FUNDAMENTOS DE PROGRAMACIÓN

LIBRO DE PROBLEMAS

Luis Joyanes Aguilar
Luis Rodríguez Baena
Matilde Fernández Azuela



Contenido



Algoritmos y programas

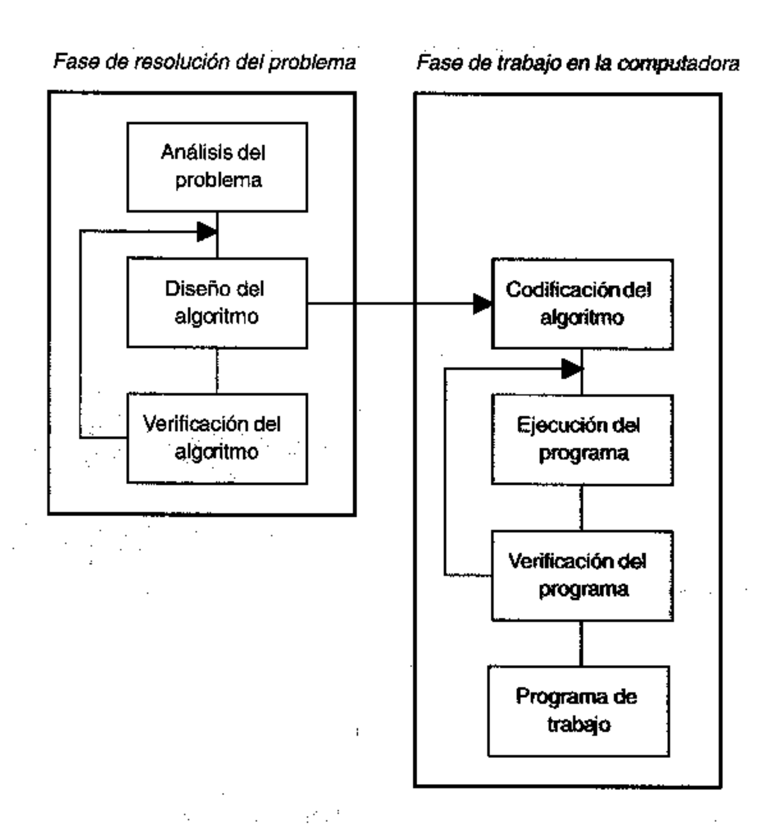
1.1. Resolución de problemas por computadoras

La principal razón para que las personas aprendan lenguajes de programación es utilizar la computadora como una herramienta para la resolución de problemas. Dos fases pueden ser identificadas en el proceso de resolución de problemas ayudados por computadora:

- Fase de resolución del problema.
- Fase de implementación (realización) en un lenguaje de programación.

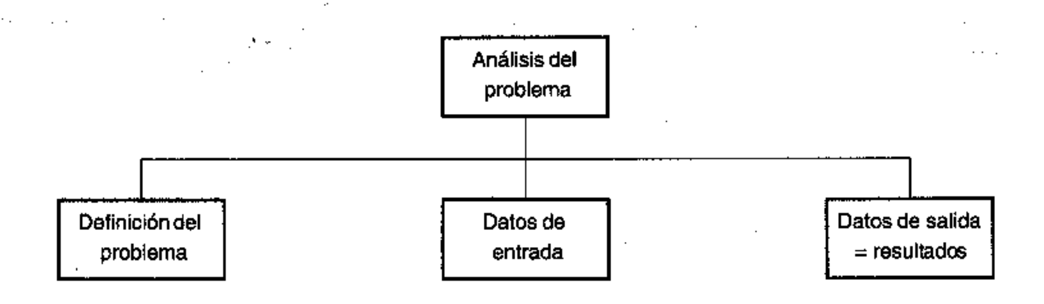
1.1.1. Fase de resolución del problema

Esta fase incluye, a su vez, el análisis del problema así como el diseño y posterior verificación del algoritmo.



1.1.1.1. Análisis del problema

El primer paso para encontrar la solución a un problema es el análisis del mismo. Se debe examinar cuidadosamente el problema a fin de obtener una idea clara sobre lo que se solicita y determinar los datos necesarios para conseguirlo.



1.1.1.2. Diseño del algoritmo

La palabra algoritmo deriva del nombre del famoso matemático y astrónomo árabe Al-Khôwarizmi (siglo IX) que escribió un conocido tratado sobre la manipulación de números y ecuaciones titulado Kitab al-jabr w'almugabala.

Un algoritmo puede ser definido como la secuencia ordenada de pasos, sin ambigüedades, que conducen a la solución de un problema dado y expresado en lenguaje natural, por ejemplo el castellano.

Todo algoritmo debe ser:

- **Preciso**. Indicando el orden de realización de cada uno de los pasos.
- **Definido.** Si se sigue el algoritmo varias veces proporcionándole los mismos datos, se deben obtener siempre los mismos resultados.
- **Finito**. Al seguir el algoritmo, éste debe terminar en algún momento, es decir tener un número finito de pasos.

Para diseñar un algoritmo se debe comenzar por identificar las tareas más importantes para resolver el problema y disponerlas en el orden en el que han de ser ejecutadas. Los pasos en esta primera descripción de actividades deberán ser refinados, añadiendo más detalles a los mismos e incluso, algunos de ellos, pueden requerir un refinamiento adicional antes de que podamos obtener un algoritmo claro, preciso y completo. Este método de diseño de los algoritmos en etapas, yendo de los conceptos generales a los de detalle a través de refinamientos sucesivos, se conoce como método descendente (top-down). En un algoritmo se deben de considerar tres partes:

- **Entrada.** Información dada al algoritmo.
- **Proceso.** Operaciones o cálculos necesarios para encontrar la solución del problema.
- **Salida**. Respuestas dadas por el algoritmo o resultados finales de los cálculos.

Como ejemplo imagine que desea desarrollar un algoritmo que calcule la superficie de un rectángulo proporcionándole su base y altura. Lo primero que deberá hacer es plantearse y contestar a las siguientes preguntas:

Especificaciones de entrada

¿Qué datos son de entrada?

¿Cuántos datos se introducirán?

¿Cuántos son datos de entrada válidos?

Especificaciones de salida

¿Cuáles son los datos de salida?

¿Cuántos datos de salida se producirán?

¿Qué precisión tendrán los resultados?

¿Se debe imprimir una cabecera?

El algoritmo en el primer diseño se podrá representar con los siguientes pasos:

Paso 1. Entrada desde periférico de entrada, por ejemplo teclado, de base y altura.
Paso 2. Cálculo de la superficie, multiplicando la base por la altura.
Paso 3. Salida por pantalla de base, altura y superficie.

El lenguaje algorítmico debe ser independiente de cualquier lenguaje de programación particular, pero fácilmente traducible a cada uno de ellos. Alcanzar estos objetivos conducirá al empleo de métodos normalizados para la representación de algoritmos, tales como los diagrama de flujo, diagrama Nassi-Schneiderman o pseudocódigo, comentados más adelante.

1.1.1.3. Verificación de algoritmos

Una vez que se ha terminado de escribir un algoritmo es necesario comprobar que realiza las tareas para las que se ha diseñado y produce el resultado correcto y esperado.

El modo más normal de comprobar un algoritmo es mediante su ejecución manual, usando datos significativos que abarquen todo el posible rango de valores y anotando en una hoja de papel las modificaciones que se producen en las diferentes fases hasta la obtención de los resultados. Este proceso se conoce como prueba del algoritmo.

1.1.2. Fase de implementación

Una vez que el algoritmo está diseñado, representado gráficamente mediante una herramienta de programación (diagrama de flujo, diagrama N-S o pseudocódigo) y verificado se debe pasar a la fase de codificación, traducción del algoritmo a un determinado lenguaje de programación, que deberá ser completada con la ejecución y comprobación del programa en el ordenador.

1.2. Datos

Dato es la expresión general que describe los objetos con los cuales opera el algoritmo. Los datos podrán ser de los siguientes tipos:

- **`entero`**. Subconjunto finito de los números enteros, cuyo rango o tamaño dependerá del lenguaje en el que posteriormente codifiquemos el algoritmo y de la computadora utilizada.
- **`real`**. Subconjunto de los números reales limitado no sólo en cuanto al tamaño, sino también en cuanto a la precisión.
- **`Lógico`**. Conjunto formado por los valores Verdad y Falso.
- **`Carácter`**. Conjunto finito y ordenado de los caracteres que la computadora reconoce.
- **`Cadena`**. Los datos (objetos) de este tipo contendrán una serie finita de caracteres, que podrán ser directamente traídos o enviados a/desde consola.

`entero`, `real`, `carácter`, `cadena` y `lógico` son tipos predefinidos en la mayoría de los lenguajes de programación. En los algoritmos para indicar que un dato es de uno de estos tipos se

declarará así, utilizando directamente el identificador o nombre del tipo. Además el usuario podrá definir sus propios tipos de datos.

Lo usual es que se definan nuevos tipos de datos agrupando valores de otros tipos definidos previamente o de tipos estándar. Por este motivo se dice que están estructurados. Si todos los valores agrupados fueran del mismo tipo a éste se le denominaría tipo base. Al establecer un tipo para un dato hay que considerar las operaciones que vamos a realizar con él y los instrumentos disponibles. Al establecer el tipo se determina la forma de representación en memoria. Los datos pueden venir expresados como constantes, variables, expresiones o funciones.

1.2.1. Constantes

Son datos cuyo valor no cambia durante todo el desarrollo del algoritmo. Las constantes podrán ser literales o con nombres.

Las constantes simbólicas o con nombre se identifican por su nombre y el valor asignado. Una constante literal es un valor de cualquier tipo que se utiliza como tal. Tendremos pues constantes:

- **Numéricas enteras.** En el rango de los enteros. Compuestas por el signo (+,-) seguido de una serie de dígitos (0..9).
- **Numéricas reales**. Compuestas por el signo (+,-) seguido de una serie de dígitos (0..9) y un punto decimal (.) o compuestas por el signo (+,-), una serie de dígitos (0..9) y un punto decimal que constituyen la mantisa, la letra E antes del exponente, el signo (+,-) y otra serie de dígitos (0..9).
- **Lógicas**. Sólo existen dos constantes lógicas, verdad y falso
- **Carácter.** Cualquier carácter del juego de caracteres utilizado colocado entre comillas simples o apóstrofes. Los caracteres que reconocen las computadoras son dígitos, caracteres alfabéticos, tanto mayúsculas como minúsculas, y caracteres especiales.
- **Cadena**. Serie de caracteres válidos encerrados entre comillas simples.

1.2.2. Variables

Una variable es un objeto cuyo valor puede cambiar durante el desarrollo del algoritmo. Se identifica por su nombre y por su tipo, que podrá ser cualquiera, y es el que determina el conjunto de valores que podrá tomar la variable. En los algoritmos se deben declarar las variables. Cuando se traduce el algoritmo a un lenguaje de programación y se ejecuta el programa resultante, la declaración de cada una de las variables originará que se reserve un determinado espacio en memoria etiquetado con el correspondiente identificador.

1.2.3. Expresiones

Una expresión es una combinación de operadores y operandos. Los operandos podrán ser constantes,

variables u otras expresiones y los operadores de cadena, aritméticos, relacionales o lógicos. Las expresiones se clasifican, según el resultado que producen, en:

- **Numéricas**. Los operandos que intervienen en ellas son numéricos, el resultado es también de tipo numérico y se construyen mediante los operadores aritméticos. Se pueden considerar análogas a las fórmulas matemáticas.
  Debido a que son los que se encuentran en la mayor parte de los lenguajes de programación, los algoritmos utilizarán los siguientes operadores aritméticos: menos unario (-), multiplicación (*), división real(/), exponenciación (^), adición (+), resta (-), módulo de la división entera (`mod`) y cociente de la división entera (`div`). Tenga en cuenta que la división real siempre dará un resultado real y que los operadores `mod` y `div` sólo operan con enteros y el resultado es entero.
- **Alfanuméricas.** Los operandos son de tipo alfanumérico y producen resultados también de dicho tipo. Se construyen mediante el operador de concatenación, representado por el operador *ampersand* (&) o con el mismo símbolo utilizado en las expresiones aritméticas para la suma.
- **Booleanas**. Su resultado podrá ser `verdad` o `falso`. Se construyen mediante los operadores relacionales y lógicos. Los operadores de relación son: igual (=), distinto (<>), menor que (<), mayor que (>), mayor o igual (>=), menor o igual (<=). Actúan sobre operandos del mismo tipo y siempre devuelven un resultado de tipo lógico. Los operadores lógicos básicos son: negación lógica (`no`), multiplicación lógica (`y`), suma lógica (o). Actúan sobre operandos de tipo lógico y devuelven resultados del mismo tipo, determinados por las tablas de verdad correspondientes a cada uno de ellos.

a	*b*	*no a*	*a y b*	*a o b*
verdad	verdad	falso	verdad	verdad
verdad	falso	falso	falso	verdad
falso	verdad	verdad	falso	verdad
falso	falso	verdad	falso	falso

El orden de prioridad general adoptado, no común a todos los lenguajes, es el siguiente:

^	Exponenciación
`no`, -	Operadores unarios
*, /, `div`, `mod`, y	Operadores multiplicativos
+, -, o	Operadores aditivos
=, <>, >, <, >=, <=	Operadores de relación

La evaluación de operadores con la misma prioridad se realizará siempre de izquierda a derecha. Si una expresión contiene subexpresiones encerradas entre paréntesis, dichas subexpresiones se evaluarán primero.

1.2.4. Funciones

En los lenguajes de programación existen ciertas funciones predefinidas o internas que aceptan unos argumentos y producen un valor denominado resultado. Como funciones numéricas, se usarán:

Funcion	*Descripción*	*Tipo de argumento*	*Resultado*
abs(x)	valor absoluto de x	entero o real	igual que el argumento
arctan(x)	arcotangente de x	entero o real	real
cos(x)	cosenode x	entero o real	real
cuadrado(x)	cuadrado de x	entero o real	igual que el argumento
ent(x)	entero de x	real	entero
exp(x)	e elevado a x	entero o real	real
ln(x)	logaritmo neperioano de x	entero o real	real
log10(x)	logaritmo base 10 de x	entero o real	real
raíz2(x)	ráiz cuadrada de x	entero de x	real
redondeo(x)	redondea x al entero más proximo	real	entero
sen(x)	seno de x	entero o real	real
trunc(x)	parte entera de x	real	entero

Las funciones se utilizarán escribiendo su nombre, seguido de los argumentos adecuados encerrados entre paréntesis, en una expresión.

1.2.5. Reglas para la construcción de identificadores

Identificadores son los nombres que se dan a las constantes simbólicas, variables, funciones, procedimientos, u otros objetos que manipula el algoritmo. La regla para construir un identificador establece que:

- debe resultar significativo, sugiriendo lo que representa.
- no podrá coincidir con palabras reservadas, propias del lenguaje algorítmico. Como se verá más adelante, la representación de algoritmos mediante pseudocódigo va a requerir la utilización de palabras reservadas.
- se admitirá un máximo de 32 caracteres.
- comenzará siempre por un carácter alfabético y los siguientes podrán ser letras, dígitos o el

símbolo de subrayado.
- podrá ser utilizado indistintamente escrito en mayúsculas o en minúsculas.

1.3. Ejercicios resueltos

Desarrolle los algoritmos que resuelvan los siguientes problemas:

1.1. Ir al cine.

Análisis del problema

DATOS DE SALIDA:	Ver la película
DATOS DE ENTRADA:	Nombre de la película, dirección de la sala, hora de proyección
DATOS AUXILIARES:	Entrada, número de asiento

Para solucionar el problema, se debe seleccionar una película de la cartelera del periódico, ir a la sala y comprar la entrada para, finalmente, poder ver la película.

Diseño del algoritmo

```
inicio
   //seleccionar la película
   tomar el periódico
   mientras no llegemos a la cartelera
      pasar la hoja
   mientras no se acabe la cartelera
      leer película
      si nos gusta, recordarla
   elegir una de las películas seleccionadas

   leer la dirección de la sala y la hora de proyección

   //comprar la entrada
   trasladarse a la sala
   si no hay entradas, ir a fin
   si hay cola
      ponerse el último
      mientras no lleguemos a la taquilla
         avanzar
         si no hay entradas, ir a fin
   comprar la entrada

   //ver la película
   leer el número de asiento de la entrada
```

```
   buscar el asiento
   sentarse
   ver la película
fin
```

1.2. Comprar una entrada para ir a los toros.

Análisis del problema

DATOS DE SALIDA:	La entrada
DATOS DE ENTRADA:	Tipo de entrada (sol, sombra, tendido, andanada,...)
DATOS AUXILIARES:	Disponibilidad de la entrada

Hay que ir a la taquilla y elegir la entrada deseada. Si hay entradas se compra (en taquilla o a los reventas). Si no la hay, se puede seleccionar otro tipo de entrada o desistir, repitiendo esta acción hasta que se ha conseguido la entrada o el posible comprador ha desistido.

Diseño del algoritmo

```
inicio
   ir a la taquilla
   si no hay entradas en taquilla
      si nos interesa comprarla en la reventa
         ir a comprar la entrada
      si no ir a fin

   //comprar la entrada
   seleccionar sol o sombra
   seleccionar barrera, tendido, andanada o palco
   seleccionar número de asiento
   solicitar la entrada
   si la tienen disponible
      adquirir la entrada
      si no
         si queremos otro tipo de entrada
            ir a comprar la entrada
fin
```

1.3. Poner la mesa para la comida.

Análisis del problema

DATOS DE SALIDA: La mesa puesta

DATOS DE ENTRADA: La vajilla, los vasos, los cubiertos, la servilletas, el número de comensales
DATOS AUXILIARES: Número de platos, vasos, cubiertos o servilletas que llevamos puestos

Para poner la mesa, después de poner el mantel, se toman las servilletas hasta que su número coincide con el de comensales y se colocan. La operación se repetirá con los vasos, platos y cubiertos.

Diseño del algoritmo

```
inicio
   poner el mantel
repetir
   tomar una servilleta
hasta que el número de servilletas es igual al de comensales

repetir
   tomar un vaso
hasta que el número de vasos es igual al de comensales

repetir
   tomar un juego de platos
hasta que el número de juegos es igual al de comensales

repetir
   tomar un juego de cubiertos
hasta que el número de juegos es igual al de comensales

fin
```

1.4. *Hacer una taza de té.*

Análisis del problema

DATOS DE SALIDA: taza de té
DATOS DE ENTRADA: bolsa de té, agua
DATOS AUXILIARES: pitido de la tetera, aspecto de la infusión

Después de echar agua en la tetera, se pone al fuego y se espera a que el agua hierva (hasta que suena el pitido de la tetera). Introducimos el té y se deja un tiempo hasta que está hecho.

Diseño del algoritmo

```
inicio
   tomar la tetera
```

```
   llenarla de agua
   encender el fuego
   poner la tetera en el fuego
   mientras no hierva el agua
      esperar
   tomar la bolsa de té
   introducirla en la tetera
   mientras no esté hecho el té
      esperar
   echar el té en la taza
fin
```

1.5. *Fregar los platos de la comida.*

Análisis del problema

DATOS DE SALIDA: platos limpios
DATOS DE ENTRADA: platos sucios
DATOS AUXILIARES: número de platos que quedan

Diseño del algoritmo

```
inicio
   abrir el grifo
   tomar el estropajo
   echarle jabón
   mientras queden platos
      lavar el plato
      aclararlo
      dejarlo en el escurridor
   mientras queden platos en el escurridor
      secar plato
fin
```

1.6. *Reparar un pinchazo de una bicicleta.*

Análisis del problema

DATOS DE SALIDA: la rueda reparada
DATOS DE ENTRADA: la rueda pinchada, los parches, el pegamento
DATOS AUXILIARES: las burbujas que salen donde está el pinchazo

Después de desmontar la rueda y la cubierta e inflar la cámara, se introduce la cámara por secciones en

un cubo de agua. Las burbujas de aire indicarán donde está el pinchazo. Una vez descubierto el pinchazo se aplica el pegamento y ponemos el parche. Finalmente se montan la cámara, la cubierta y la rueda.

Diseño del algoritmo

```
inicio
   desmontar la rueda
   desmontar la cubierta
   sacar la cámara
   inflar la cámara
   meter una sección de la cámara en un cubo de agua
   mientras no salgan burbujas
      meter una sección de la cámara en un cubo de agua
   marcar el pinchazo
   echar pegamento
   mientras no esté seco
      esperar
   poner el parche
   mientras no esté fijo
      apretar
   montar la cámara
   montar la cubierta
   montar la rueda
fin
```

1.7. Pagar una multa de tráfico.

Análisis del problema

DATOS DE SALIDA:	el recibo de haber pagado la multa
DATOS DE ENTRADA:	datos de la multa
DATOS AUXILIARES:	impreso de ingreso en el banco, dinero

Debe elegir cómo desea pagar la multa, si en metálico en la Delegación de Tráfico o por medio de una transferencia bancaria. Si elige el primer caso, tiene que ir a la Delegación Provincial de Tráfico, desplazarse a la ventanilla de pago de multas, pagarla en efectivo y recoger el resguardo.

Si desea pagarla por transferencia bancaria, hay que ir al banco, rellenar el impreso apropiado, dirigirse a la ventanilla, entregar el impreso y recoger el resguardo.

Diseño del algoritmo

```
inicio
   si pagamos en efectivo
```

```
      ir a la Delegación de Tráfico
      ir a la ventanilla de pago de multas
      si hay cola
         ponerse el último
         mientras no lleguemos a la ventanilla
            esperar
      entregar la multa
      entregar el dinero
      recoger el recibo
   si no
   //pagamos por trasferencia bancaria
      ir al banco
      rellenar el impreso
      si hay cola
         ponerse el último
         mientras no lleguemos a la ventanilla
            esperar
      entregar el impreso
      recoger el resguardo
fin
```

1.8. *Hacer una llamada telefónica. Considerar los casos:* a) *llamada manual con operador;* b) *llamada automática ;* c) *llamada a cobro revertido.*

Análisis del problema

Para decidir el tipo de llamada que se efectuará, primero se debe considerar si se dispone de efectivo o no para realizar la llamada a cobro revertido. Si hay efectivo se debe ver si el lugar a donde vamos a llamar está conectado a la red automática o no.

Para una llamada con operadora hay que llamar a la centralita y solicitar la llamada, esperando hasta que se establezca la comunicación. Para una llamada automática se leen los prefijos del pais y provincia si fuera necesario, y se realiza la llamada, esperando hasta que cojan el teléfono. Para llamar a cobro revertido se debe llamar a centralita, solicitar la llamada y esperar a que el abonado del teléfono al que se llama désu autorización, con lo que se establecerá la comunicación.

Como datos de entrada tendríamos las variables que nos van a condicionar el tipo de llamada, el número de teléfono y, en caso de llamada automática, los prefijos si los hubiera. Como dato auxiliar se podría considerar en los casos a y c el contacto con la centralita.

Diseño del algoritmo

```
inicio
   si tenemos dinero
```

```
      si podemos hacer una llamada automática
         leer el prefijo de país y localidad
         marcar el número
      si no
         //llamada manual
         llamar a la centralita
         solicitar la comunicación
      mientras no contesten
         esperar
      establecer comunicación
   si no
      //realizar una llamada a cobro revertido
      llamar a la centralita
      solicitar la llamada
      esperar hasta tener la autorización
      establecer comunicación
fin
```

1.9. Cambiar el cristal roto de la ventana.

Análisis del problema

DATOS DE SALIDA:	la ventana con el cristal nuevo
DATOS DE ENTRADA:	el cristal nuevo
DATOS AUXILIARES:	el número de clavos de cada una de las molduras

Diseño del algoritmo

```
inicio
   repetir cuatro veces
      quitar un clavo
      mientras el número de clavos quitados no sea igual al total de clavos
         quitar un clavo
      sacar la moldura
   sacar el cristal roto
   poner el cristal nuevo
   repetir cuatro veces
      poner la moldura
      poner un clavo
      mientras el número de clavos puestos no sea igual al total de clavos
      poner un clavo
   poner el cristal nuevo
fin
```

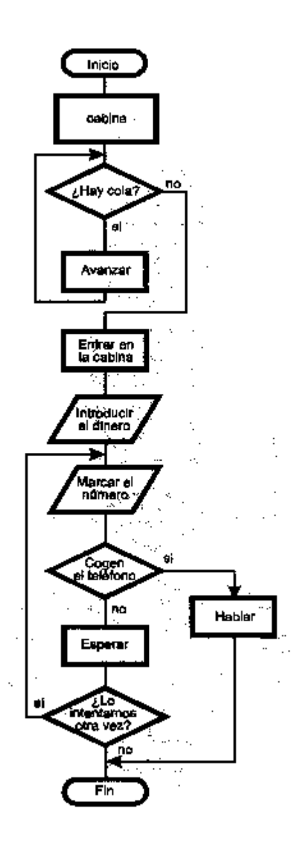
1.10. Realizar una llamada telefónica desde un teléfono público.

Análisis del problema

DATOS DE SALIDA: la comunicación por teléfono
DATOS DE ENTRADA: el número de teléfono, el dinero
DATOS AUXILIARES: distintas señales de la llamada (comunicando, etc.)

Se debe ir a la cabina y esperar si hay cola. Entrar e introducir el dinero. Se marca el número y se espera la señal, si está comunicando o no contestan se repite la operación hasta que descuelgan el teléfono o decide irse.

Diseño del algoritmo



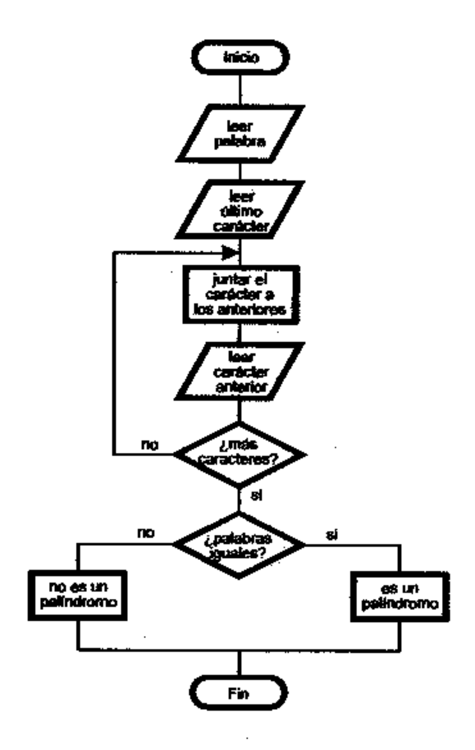
1.11. Averiguar si una palabra es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda, como por ejemplo, 'radar'.

Análisis del problema

DATOS DE SALIDA:	el mensaje que nos dice si es o no un palíndromo
DATOS DE ENTRADA:	palabra
DATOS AUXILIARES:	cada carácter de la palabra, palabra al revés

Para comprobar si una palabra es un palíndromo, se puede ir formando una palabra con los caracteres invertidos con respecto a la original y comprobar si la palabra al revés es igual a la original. Para obtener esa palabra al revés, se leerán en sentido inverso los caracteres de la palabra inicial y se irán juntando sucesivamente hasta llegar al primer carácter.

Diseño del algoritmo



1.12. Escribir un algoritmo para determinar el máximo común divisor de dos números enteros por el algoritmo de Euclides.

Análisis del problema

DATOS DE SALIDA: máximo común divisor (`mcd`)
DATOS DE ENTRADA: dos números enteros (`a` y `b`)
DATOS AUXILIARES: `resto`

Para hallar el máximo común divisor de dos números se debe dividir uno entre otro. Si la división es exacta, es decir si el resto es 0, el máximo común divisor es el divisor. Si no, se deben dividir otra vez los números, pero en este caso el dividendo será el antiguo divisor y el divisor el resto de la división anterior. El proceso se repetirá hasta que la división sea exacta.

Para diseñar el algoritmo se debe crear un bucle que se repita mientras que la división no sea exacta. Dentro del bucle se asignarán nuevos valores al dividendo y al divisor.

Diseño del algoritmo

```
inicio
  leer (a,b)
  mientras a mod b <> 0
    resto ← a mod b
    a ← b
    b ← resto
  mcd ← b
  escribir (mcd)
fin
```

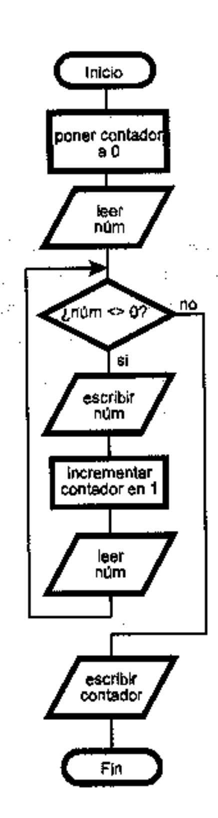
1.13. Diseñar un algoritmo que lea e imprima una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe imprimir. Finalmente se desea obtener la cantidad de valores leídos distintos de 0.

Análisis del problema

DATOS DE ENTRADA: los distintos números (`núm`)
DATOS DE SALIDA: los mismos números menos el 0, la cantidad de números (`contador`)

Se deben leer números dentro de un bucle que terminará cuando el último número leído sea cero. Cada vez que se ejecute dicho bucle y antes que se lea el siguiente número se imprime éste y se incrementa el contador en una unidad. Una vez se haya salido del bucle se debe escribir la cantidad de números leídos, es decir, el contador.

Diseño del algoritmo



1.14. Diseñar un algoritmo que imprima y sume la serie de números 3,6,9,12,...,99.

Análisis del problema

Se trata de idear un método con el que obtengamos dicha serie, que no es más que incrementar una variable de tres en tres. Para ello se hará un bucle que se acabe cuando el número sea mayor que 99 (o cuando se realice 33 veces). Dentro de ese bucle se incrementa la variable, se imprime y se acumula su valor en otra variable llamada `suma`, que será el dato de salida.

No tendremos por tanto ninguna variable de entrada, y sí dos de salida, la que nos va sacando los números de tres en tres (`núm`) y `suma`.

Diseño del algoritmo

```
inicio
   suma ← 0
   núm ← 3
   mientras núm <= 99 hacer
      escribir (núm)
      suma ← suma + num
      núm ← núm + 3
   escribir (suma)
fin
```

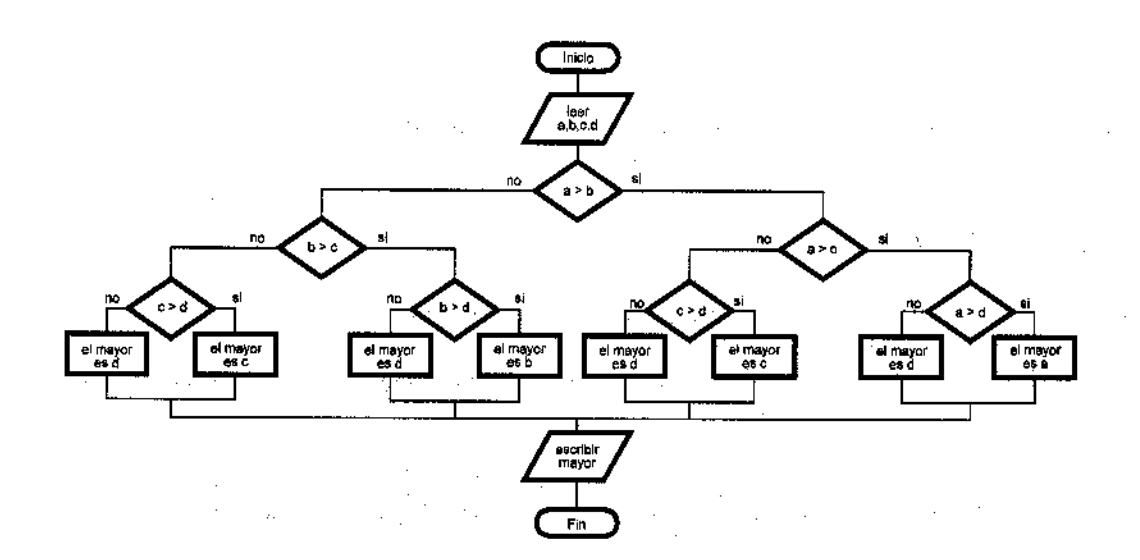
1.15. *Escribir un algoritmo que lea cuatro números y, a continuación, escriba el mayor de los cuatro.*

Análisis del problema

DATOS DE SALIDA: mayor (el mayor de los cuatro números)
DATOS DE ENTRADA: a, b, c, d (los números que leemos por teclado)

Hay que comparar los cuatro números, pero no hay necesidad de compararlos todos con todos. Por ejemplo, si a es mayor que b y a es mayor que c, es evidente que ni b ni c son los mayores, por lo que si a es mayor que d el número mayor será a y en caso contrario lo será d.

Diseño del algoritmo



1.16. Diseñar un algoritmo para calcular la velocidad (en metros/segundo) de los corredores de una carrera de 1500 metros. La entrada serán parejas de números (minutos, segundos) que darán el tiempo de cada corredor. Por cada corredor se imprimirá el tiempo en minutos y segundos, así como la velocidad media. El bucle se ejecutará hasta que demos una entrada de 0,0 que será la marca de fin de entrada de datos.

Análisis del problema

DATOS DE SALIDA: `v` (velocidad media)
DATOS DE ENTRADA: `mm,ss` (minutos y segundos)
DATOS AUXILIARES: `distancia` (distancia recorrida, que en el ejemplo es de 1500 metros) y `tiempo` (los minutos y los segundos que ha tardado en recorrerla)

Se debe efectuar un bucle que se ejecute hasta que mm sea 0 y ss sea 0. Dentro del bucle se calcula el tiempo en segundos con la fórmula tiempo = ss + mm * 60. La velocidad se hallará con la fórmula `velocidad = distancia /tiempo`.

Diseño del algoritmo

```
inicio
   distancia ← 1500
   leer (mm,ss)
   mientras mm = 0 y ss = 0 hacer
      tiempo ← ss + mm * 60
      v ← distancia / tiempo
      escribir (mm,ss,v)
      leer (mm,ss)
fin
```

1.17. Diseñar un algoritmo para determinar si un número n es primo. (un número primo sólo es divisible por el mismo y por la unidad).

Análisis del problema

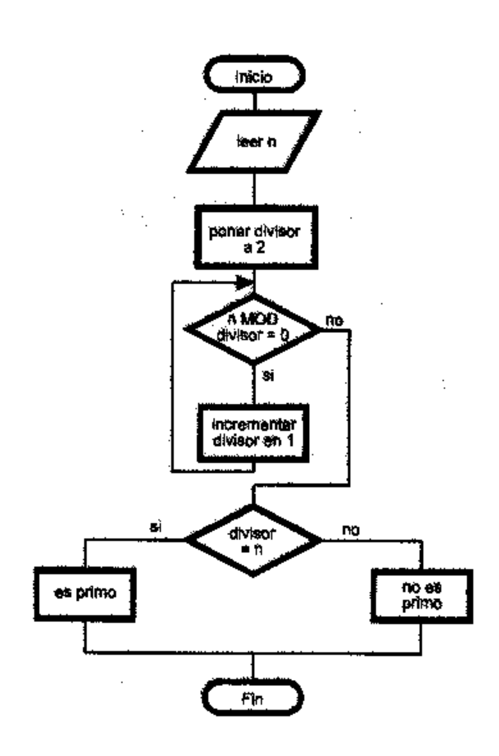
DATOS DE SALIDA: El mensaje que nos indica si es o no primo
DATOS DE ENTRADA: `n`
DATOS AUXILIARES: `divisor` (es el número por el que vamos a dividir `n` para averiguar si es primo)

Una forma de averiguar si un número es primo es por tanteo. Para ello se divide sucesivamente el número por los números comprendidos entre 2 y `n`. Si antes de llegar a `n` encuentra un divisor exacto,

el número no será primo. Si el primer divisor es n el número será primo.

Por lo tanto se hará un bucle en el que una variable (`divisor`) irá incrementándose en una unidad entre 2 y n. El bucle se ejecutará hasta que se encuentra un divisor, es decir hasta que `n mod divisor = 0`. Si al salir del bucle `divisor = n`, el número será primo.

Diseño del algoritmo



1.18. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura.

Análisis del problema

DATOS DE SALIDA: `s` (superficie)
DATOS DE ENTRADA: `b` (base), `a` (altura)

Para calcular la superficie se aplica la fórmula `S = base * altura /2.`

Diseño del algoritmo

```
inicio
   leer (b,a)
   s ← b * a / 2
   escribir (s)
fin
```



La resolución de problemas con computadoras y las herramientas de programación

2.1. Herramientas de programación

Un algoritmo puede ser escrito en castellano narrativo, pero esta descripción suele ser demasiado prolija y, además, ambigua. Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo de los lenguajes de programación y, al mismo tiempo, conseguir que sea fácilmente codificable.

Los métodos más usuales para la representación de algoritmos son:

A. Diagrama de flujo

B. Diagrama N-S (Nassi-Schneiderman)

C. Pseudocódigo

2.2. Diagrama de flujo

Los diagramas de flujo se utilizan tanto para la representación gráfica de las operaciones ejecutadas sobre los datos a través de todas las partes de un sistema de procesamiento de información, diagrama de flujo del sistema, como para la representación de la secuencia de pasos necesarios para describir un procedimiento particular, diagrama de flujo de detalle.

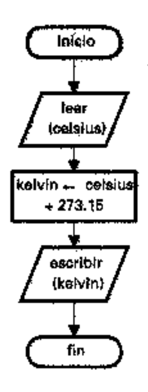
En la actualidad se siguen usando los diagramas de flujo del sistema, pero ha decaído el uso de los diagramas de flujo de detalle al aparecer otros métodos de diseño estructurado más eficaces para la representación y actualización de los algoritmos.

Los diagramas de flujo de detalle son, no obstante, uno de nuestros objetivos prioritarios y, a partir de ahora, los denominaremos simplemente diagramas de flujo.

El diagrama de flujo utiliza unos símbolos normalizados, con los pasos del algoritmo escritos en el símbolo adecuado y los símbolos unidos por flechas, denominadas líneas de flujo, que indican el orden en que los pasos deben ser ejecutados. Los símbolos principales son:

Símbolo	***Función***
	Inicio y fin del algoritmo
	Proceso
	Entrada / Salida
	Decisión
	Comentario

Resulta necesario indicar dentro de los símbolos la operación específica concebida por el programador. Como ejemplo veamos un diagrama de flujo básico, que representa la secuencia de pasos necesaria para que un programa lea una temperatura en grados Centígrados y calcule y escriba su valor en grados Kelvin.



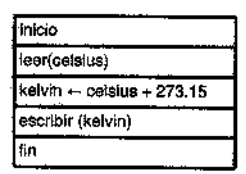
2.3. Diagrama Nassi-Schneiderman

Los diagramas Nassi-Schneiderman, denominados así por sus inventores, o también N-S, o de Chapin son una herramienta de programación que favorece la programación estructurada y reúne característi-cas gráficas propias de diagramas de flujo y lingüísticas propias de los pseudocódigos. Constan de una serie de cajas contiguas que se leerán siempre de arriba-abajo y se documentarán de la forma adecuada.

En los diagramas N-S las tres estructuras básicas de la programación estructurada, secuenciales, selectivas y repetitivas, encuentran su representación propia. La programación estucturada será tratada en capítulos posteriores.

Símbolo	***Tipo de estructura***
sentencia 1 / sentencia 2 / sentencia n	Secuencial
	Repetitiva de 0 a n veces
	Repetitiva de 1 a n veces
	Repetitiva n veces
condición / sí / no	Selectiva

El algoritmo que lee una temperatura en grados Celsius y calcula y escribe su valor en grados Kelvin se puede representar mediante el siguiente diagrama N-S:



2.4. Pseudocódigo

El pseudocódigo es un lenguaje de especificación de algoritmos que utiliza palabras reservadas y exige la indentación, o sea sangría en el margen izquierdo, de algunas líneas. En nuestros pseudocódigos usaremos determinadas palabras en español como palabras reservadas.

El pseudocódigo se concibió para superar las dos principales desventajas del diagrama de flujo:

lento de crear y difícil de modificar sin un nuevo redibujo. Es una herramienta muy buena para el seguimiento de la lógica de un algoritmo y para transformar con facilidad los algoritmos a programas, escritos en un lenguaje de programación específico. En este libro se escribirán casi todos los algoritmos en pseudocódigo.

El pseudocódigo comenzará siempre con la palabra **inicio** y terminará con la palabra **fin**. Cuando se coloque un comentario de una sola línea se escribirá precedido de //. Si el comentario es multilínea, lo pondremos entre {}. Para introducir un valor o serie de valores desde el dispositivo estándar y almacenarlos en una o varias variables utilizaremos **leer**(*<lista_de_variables>*).

Con *<nombre_de_variable>* ← *<expresión>* almacenaremos en una variable el resultado de evaluar una expresión. Hay que tener en cuenta que una única constante, variable o función, constituyen una expresión.

Para imprimir en el dispositivo estándar de salida una o varias expresiones emplearemos **escribir**(*<lista_de_expresiones>*)

Las distintas estructuras se representarán de la siguiente forma:

Decisión simple:

```
si <condición> entonces
   <acciones1>
fin_si
```

Decisión doble:

```
si <condición> entonces
   <acciones1>
si_no
   <acciones2>
fin_si
```

Decisión múltiple:

```
según_sea <expresión_ordinal> hacer
   <lista_de_valores_ordinales>: <acciones1>
   .......
[si_no       // El corchete indica opcionalidad
   <accionesN>]
fin_según
```

Repetitivas:

```
mientras <condición> hacer
   <acciones>
fin_mientras
repetir
   <acciones>
hasta_que <condición>
desde <variable>←<v_inicial> hasta <v_final>
         [incremento | decremento <incremento>] hacer
         <acciones>
fin_desde
```

inicio, **fin**, **leer**, **escribir** y las palabras que aparecen en negrita en las distintas estructuras se consideran palabras reservadas y no deberán utilizarse en su lugar otras similares.

El ejemplo ya citado que transforma grados Celsius en Kelvin, escrito en pseudocódigo quedaría de la siguiente forma:

```
inicio
  leer(celsius)
  kelvin ← celsius + 273.15
  escribir(Kelvin)
fin
```

2.5. Ejercicios resueltos

2.1. *¿Cuál de los siguientes datos son válidos para procesar por una computadora?*

a) 3.14159	*e) 2.234E2*	*i) 12.5E.3*
b) 0.0014	*f) 12E+6*	*j) .123E4*
c) 12345.0	*g) 1.1E-3*	*k) 5A4.14*
d) 15.0E-04	*h) -15E-0.4*	*l) A1.E04*

Serían válidos los datos a, b, c, d, e, f, g y j. Los datos h e i no serían válidos pues el exponente no puede tener forma decimal. El k, no sería correcto pues mezcla caracteres alfabéticos y dígitos, y no se puede considerar una identificador de una variable ya que empieza por un dígito. El dato l aunque mezcla dígitos y caracteres alfabéticos, podría ser un identificador, si admitiéramos el carácter punto como carácter válido para una variable (PASCAL o BASIC lo consideran).

2.2. *¿Cuál de los siguientes identificadores son válidos?*

a) Renta	*e) Dos Pulgadas*	*i) 4A2D2*
b) Alquiler	*f) C3PO*	*j) 13Nombre*
c) Constante	*g) Bienvenido#5*	*k) Nombre_Apellidos*
d) Tom's	*h) elemento*	*l) NombreApellidos*

Se consideran correctos los identificadores a, b, c, f, h, k y l. El d no se considera correcto pues incluye el apóstrofo que no es un carácter válido en un identificador. Lo mismo ocurre con el e y el espacio en blanco, y el g con el carácter #. El i y el j no serían válidos al no comenzar por un carácter alfabético.

2.3. *Escribir un algoritmo que lea un valor entero, lo doble, se multiplique por 25 y visualice el resultado.*

Análisis de problema

DATOS DE SALIDA: `resultado` (es el resultado de realizar las operaciones)
DATOS DE ENTRADA: `número` (el número que leemos por teclado)

Leemos el número por teclado y lo multiplicamos por 2, metiendo el contenido en la propia variable de entrada. A continuación lo multiplicamos por 25 y asignamos el resultado a la variable de salida `resultado`. También podríamos asignar a la variable de salida directamente la expresión `numero*2*25`.

Diseño del algoritmo:

```
algoritmo  ejercicio_2_3
   entero : número,resultado
inicio
   leer  (número)
   número ← número * 2
   resultado ← número * 25
   escribir  (resultado)
fin
```

2.4. *Diseñar un algoritmo que lea cuatro variables y calcule e imprima su producto, su suma y su media aritmética.*

Análisis del problema

DATOS DE SALIDA: `producto`, `suma` y `media`
DATOS DE ENTRADA: `a,b,c,d`

Después de leer los cuatro datos, asignamos a la variable `producto` la multiplicación de las cuatro variables de entrada. A la variable `suma` le asignamos su suma y a la variable `media` asignamos el resultado de sumar las cuatro variables y dividirlas entre cuatro. Como el operador suma tiene menos prioridad que el operador división, la será necesario encerrar la suma entre paréntesis. También podríamos haber dividido directamente la variable `suma` entre cuatro.

La variables `a,b,c, d`, `producto` y `suma` podrán ser enteras, pero no así la variable `media`, ya que la división produce siempre resultados de tipo real.

Diseño del algoritmo

```
algoritmo  ejercicio_2__4
   entero:  a,b,c,d,producto,suma
```

```
   real: media
inicio
   leer (a,b,c,d)
   producto ← a * b * c * d
   suma ← a + b + c + d
   media ←(a + b + c + d) / 4
   escribir (producto,suma,media)
fin
```

2.5. *Diseñar un programa que lea el peso de un hombre en libras y nos devuelva su peso en kilogramos y gramos (Nota: una libra equivale a 0.453592 kilogramos)*

Análisis del problema

DATOS DE SALIDA: `kg, gr`
DATOS DE ENTRADA: `peso`
DATOS AUXILIARES: `libra` (los kilogramos que equivalen a una libra)

El dato auxiliar `libra` lo vamos a considerar como una constante, pues no variará a lo largo del programa. Primero leemos el peso. Para hallar su equivalencia en kilogramos multiplicamos éste por la constante libra. Para hallar el peso en gramos multiplicamos los kilogramos entre mil. Como es posible que el dato de entrada no sea exacto, consideraremos todas las variables como reales.

Diseño del algoritmo

```
algoritmo ejercicio_2_5
      constante libra = 0.453592
      real: peso,kg,gr
inicio
      leer (peso)
      kg ← peso * libra
      gr ← kg * 1000
      escribir ('Peso en kilogramos: ',kg)
      escribir ('Peso en gramos: ',gr)
fin
```

2.6. *Si A=6, B=2 y C=3, encontrar los valores de las siguientes expresiones:*

a) A-B+C *d) A*B **mod** C*

*b) A*B **div** C* *e) A+B **mod** C*

*c) A **div** B + C* *f) A **div** B **div** C*

Los resultados serían:

a) (6-2)+3 = 7
b) (6*2) **div** 3 = 4
c) (6 **div** 2)+3 = 6
d) (6*2) **mod** 3 = 0
e) 6+(2 **mod** 3) = 8
f) (6 **div** 2) **div** 3 = 1

2.7. *¿Qué se obtiene en las variables A y B después de la ejecución de las siguientes instrucciones?*

A ← 5

B ← A + 6

A ← A + 1

B ← A - 5

Las variables irían tomando los siguientes valores:

A ← 5	,	A = 5, B indeterminado
B ← A + 6	,	A = 5, B = 5 + 6 = 11
A ← A + 1	,	A = 5 + 1 = 6, B = 11
B ← A - 5	,	A = 6, B = 6 - 5 = 1

Los valores finales serían: A = 6 y B = 1.

2.8. *¿Qué se obtiene en las variables A, B y C después de ejecutar las siguientes instrucciones?*

A ← 3

B ← 20

C ← A + B

B ← A + B

A ← B

Las variables tomarían sucesivamente los valores:

A ← 3	, A = 3, B y C indeterminados
B ← 20	, A = 3, B = 20, C indeterminado
C ← A + B	, A = 3, B = 20, C = 3 + 20 = 23
B ← A + B	, A = 3, B = 3 + 20 = 23, C = 23
A ← B	, A = 23, B = 23, C = 23

Los valores de las variables serían: A = 23, B = 23 y C = 23.

2.9. *¿Que valor toman las variables A y B tras la ejecución de las siguientes asignaciones?*

$A \leftarrow 10$

$B \leftarrow 5$

$A \leftarrow B$

$B \leftarrow A$

El resultado sería el siguiente:

A ← 10 , A = 10, B indeterminada
B ← 5 , A = 10, B = 5
A ← B , A = 5, B = 5
B ← A , A = 5, B = 5

con lo que A y B tomarían el valor 5.

2.10. *Escribir las siguientes expresiones en forma de expresiones algorítmicas*

a) $\dfrac{M}{N} + 4$ b) $M + \dfrac{N}{P - Q}$ c) $\dfrac{\text{sen } x + \cos y}{\tan x}$

d) $\dfrac{M + N}{P - Q}$ e) $\dfrac{P + \dfrac{N}{P}}{Q - \dfrac{r}{5}}$ f) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

a) `M / N + 4`
b) `M + N /(P-Q)`
c) `(`**`sen`**`(X) + `**`cos`**`(X)) / `**`tan`**`(X)`
d) `(M + N) / (P - Q)`
e) `(M + N / P) / (Q - R / 5)`
f) `((-B) + `**`raiz2`**`(BE2-4*A*C))/(2*A)`

2.11. *Se tienen tres variables A, B y C. Escribir las instrucciones necesarias para intercambiar entre sí sus valores del modo siguiente:*

B toma el valor de A

C toma el valor de B

A toma el valor de C

Nota: sólo se debe utilizar una variable auxiliar que llamaremos AUX.

```
AUX ← A
    A ← C
    C ← B
    B ← AUX
```

2.12. Deducir los resultados que se obtienen del siguiente algoritmo:

```
algoritmo ejercicio_2_12
   entero: X, Y, Z
inicio
   X ← 15
   Y ← 30
   Z ← Y - X
   escribir (X,Y)
   escribir (Z)
fin
```

Para analizar los resultados de un algoritmo, lo mejor es utilizar una tabla de seguimiento. En la tabla de seguimiento aparecen varias columnas, cada una con el nombre de una de las variables que aparecen en el algoritmo, pudiéndose incluir además una con la salida del algoritmo. Más abajo se va realizando el seguimiento del algoritmo rellenando la columna de cada variable con el valor que ésta va tomando.

X	Y	Z	Salida
15	30	15	
			15,30
			15

2.13. Encontrar el valor de la variable VALOR después de la ejecución de las siguientes operaciones

a) $VALOR \leftarrow 4.0 * 5$

b) $X \leftarrow 3.0$

$Y \leftarrow 2.0$

$VALOR \leftarrow X \wedge Y - Y$

c) $VALOR \leftarrow 5$

$X \leftarrow 3$

$VALOR \leftarrow VALOR * X$

Los resultados serían:

a) VALOR = 20.0

b) VALOR = 7.0

c) VALOR = 15

Nótese que en los casos a) y b), `valor` tendrá un contenido de tipo real, ya que alguno de los operandos son de tipo real.

2.14. Determinar los valores de A, B, C y D después de la ejecución de las siguientes instrucciones:

```
algoritmo ejercicio_2_14
var
   entero: A, B, C, D
inicio
   A ← 1
   B ← 4
   C ← A + B
   D ← A - B
   A ← C + 2 * B
   B ← C + B
   C ← A * B
   D ← B + D
   A ← D + C
   si C > D entonces
      C ← A - D
   si_no
      C ← B - D
   fin_si
fin
```

Realizaremos una tabla de seguimiento:

A	B	C	D
1	4	5	-3
13	9	117	6
123		117	

con lo que A = 123, B = 9, C = 117 y D = 6

2.15. Escribir un algoritmo que calcule y escriba el cuadrado de 821.

Análisis del problema

DATOS DE SALIDA: `cuadr` (almacenará el cuadrado de 821)
DATOS DE ENTRADA: el propio número 821

Lo único que hará este algoritmo será asignar a la variable cuadr el cuadrado de 821, es decir, 821 * 821

Diseño del algoritmo

```
algoritmo ejercicio_2_15
   entero: cuadr
inicio
   cuadr ← 821 * 821
   escribir (cuadr)
fin
```

2.16. *Realizar un algoritmo que calcule la suma de los enteros entre 1 y 10, es decir 1+2+3+...+10*

Análisis del problema

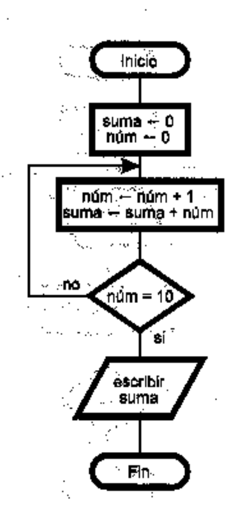
DATOS DE SALIDA:	suma (contiene la suma requerida)
DATOS AUXILIARES:	núm (será una variable que vaya tomando valores entre 1 y 10 y se acumulará en suma)

Hay que ejecutar un bucle que se realice 10 veces. En él se irá incrementando en 1 la variable núm, y se acumulará su valor en la variable suma. Una vez salgamos del bucle se visualizará el valor de la variable suma.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : suma,núm
```



2.17. *Realizar un algoritmo que calcule y visualice las potencias de 2 entre 0 y 10.*

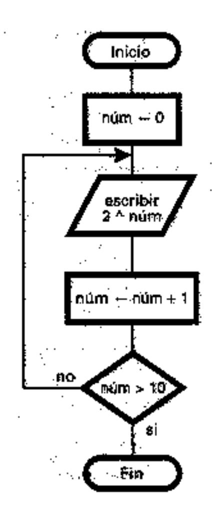
Análisis del problema

Hay que implementar un bucle que se ejecute once veces y dentro de él ir incrementando una variable que tome valores entre 0 y 10 y que se llamará `núm`. También dentro de él se visualizará el resultado de la operación `2 ^ núm`.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero: núm
```



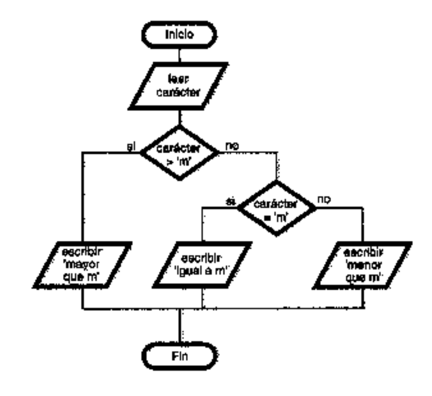
2.18. Leer un carácter y deducir si está situado antes o después de la «m» en orden alfabético.

Análisis del problema

Como dato de salida está el mensaje que nos dice la situación del carácter con respecto a la «m». Como entrada el propio carácter que introducimos por teclado. No se necesita ninguna variable auxiliar.

Se debe leer el carácter y compararlo con la «m» para ver si es mayor, menor o igual que ésta. Recuerde que para el ordenador también un carácter puede ser mayor o menor que otro, y lo hace comparando el código de los dos elementos de la comparación.

Diseño del algoritmo



2.19. Leer dos caracteres y deducir si están en orden alfabético.

Análisis del problema

DATOS DE SALIDA: El mensaje que nos indica si están en orden alfabético
DATOS DE ENTRADA: A y B (los caracteres que introducimos por teclado

Se deben leer los dos caracteres y compararlos. Si A es mayor que B, no se habrán introducido en orden. En caso contrario estarán en orden o serán iguales.

Diseño del algoritmo

```
algoritmo  ejercicio_2_19
   carácter: a, b
inicio
   leer  (a,b)
   si a < b entonces
      escribir('están en orden')
   si_no
      si a = b entonces
         escribir ('son iguales')
      si_no
         escribir ('no están en orden')
      fin_si
   fin_si
fin
```

2.20. Leer un carácter y deducir si está o no comprendido entre las letras I y M ambas inclusive.

Análisis del problema

DATOS DE SALIDA: El mensaje
DATOS DE ENTRADA: El carácter

Se lee el carácter y se comprueba si está comprendido entre ambas letras con los operadores >= y <=.

Diseño del algoritmo

```
algoritmo Ejercicio_2_20
   carácter: c
inicio
   leer (c)
```

```
   si (c >= 'I') y (c <= 'M') entonces
      escribir ('Está entre la I y la M')
   si_no
      escribir ('no se encuentra en ese rango')
   fin_si
fin
```



Estructura general de un programa

3.1. Escritura de algoritmos

En la escritura de algoritmos se hará necesario el uso de alguna herramienta de programación. La más adecuada es el pseudocódigo, y debe quedar todo lo más claro posible, de modo que se facilite al máximo su posterior codificación en un lenguaje de programación.

Al escribir un algoritmo deberemos considerar las siguientes partes:

- La cabecera
- El cuerpo del algoritmo.

La cabecera contiene el nombre del algoritmo completo precedido por la palabra **algoritmo**.

El cuerpo del algoritmo contiene a su vez otras dos secciones: el bloque de declaraciones y el bloque de instrucciones. En el bloque de declaraciones se definen o declaran las constantes con nombre, los tipos de datos definidos por el usuario y también las variables. Se recomienda seguir este orden.

Para declarar las constantes con nombre el formato será

```
const
   <nombre_de_constante1> = <valor1>
   ...............................
```

La declaración de tipos suele realizarse en base a los tipos estándar o a otros definidos previamente, aunque también hay que considerar el método directo de enumeración de los valores constituyentes. Todo esto se comentará con más detalle en capítulos posteriores.

Al declarar las variables tendremos que listar sus nombres y especificar sus tipos, lo que haremos de la siguiente forma:

```
var
   <tipo_de_dato1>: <lista_de_variables>
   ...................................
```

El bloque de instrucciones contiene las acciones a ejecutar para la obtención de los resultados. Las instrucciones o acciones básicas a colocar en este bloque se podrían clasificar del siguiente modo:

- **de inicio/fin**. La primera instrucción de este bloque será siempre la de inicio y la última la de fin.
- **de asignación**. Esta instrucción permite almacenar en una variable el resultado de evaluar una expresión, perdiéndose cualquier otro valor previo que la variable pudiera tener. Su formato es:

  ```
  <nombre_de_variable> ← <expresión>
  ```

  Una expresión puede estar formada por una única constante, variable o función. La variable que recibe el valor final de la expresión puede intervenir en la misma, con lo que se da origen a contadores y acumuladores.
- **de lectura**. Toma uno o varios valores desde un dispositivo de entrada y los almacena en memoria en las variables que aparecen listadas en la propia instrucción. Cuando el dispositivo sea el dispositivo estándar de entrada, escribiremos:

  ```
  leer(<lista_de_variables>)
  ```
- **de escritura**. Envía datos a un dispositivo. Cuando los datos los enviemos al dispositivo estándar el formato para la instrucción será:

  ```
  escribir(<lista_de_expresiones>)
  ```
- **de bifurcación**. Estas instrucciones no realizan trabajo efectivo alguno, pero permiten controlar el que se ejecuten o no otras instrucciones, así como alterar el orden en el que las acciones son ejecutadas. Las bifurcaciones en el flujo de un programa se realizarán de modo condicional, esto es en función del resultado de la evaluación de una condición.
  El desarrollo lineal de un programa se interrumpe con este tipo de instrucciones y, según el punto a donde se bifurca, podremos clasificarlas en bifurcaciones hacia adelante o hacia atrás. Las representaremos mediante estructuras selectivas o repetitivas.

Además, resulta recomendable que los algoritmos lleven comentarios.

3.2. Contadores, acumuladores e interruptores

Entre las variables que utilizan los algoritmos merecen una especial mención contadores, acumuladores e interruptores.

3.2.1. Contadores

Un contador es una variable cuyo valor se incrementa o decrementa en una cantidad constante cada vez que se produce un determinado suceso o acción. Los contadores se utilizan en las estructuras repetitivas

con la finalidad de contar sucesos o acciones internas del bucle.

Con los contadores deberemos realizar una operación de inicialización y, posteriormente, las sucesivas de incremento o decremento del contador.

La inicialización consiste en asignarle al contador un valor. Se situará antes y fuera del bucle.

```
<nombre_del_contador> ← <valor_de_inicialización>
```

En cuanto a los incrementos o decrementos del contador, puesto que la operación de asignación admite que la variable que recibe el valor final de una expresión intervenga en la misma, se realizarán a través de este tipo de instrucciones de asignación, de la siguiente forma:

```
<nombre_del_contador> ← <nombre_del_contador> + <valor_constante>
```

dicho `<valor_constante>` podrá ser positivo o negativo. Esta instrucción se colocará en el interior del bucle.

3.2.2. Acumuladores

Son variables cuyo valor se incrementa o decrementa en una cantidad variable. Necesitan operaciones de:

- Inicialización

  ```
  <nombre_acumulador> ← <valor_de_inicialización>
  ```

- Acumulación

  ```
  <nombre_acumulador> ← <nombre_acumulador> + <nombre_variable>
  ```

  Hay que tener en cuenta que la siguiente también sería una operación de acumulación:

  ```
  <nombre_acumulador> ← <nombre_acumulador> * <valor>
  ```

3.2.3. Interruptores

Un interruptor, bandera o *switch* es una variable que puede tomar los valores **verdad** y **falso** a lo largo de la ejecución de un programa, comunicando así información de una parte a otra del mismo. Pueden ser utilizados para el control de bucles.

3.3. Estilo recomendado para la escritura de algoritmos

Como se dijo anteriormente, la herramienta de escritura de algoritmos más adecuada es el pseudocódigo. Con el empleo del pseudocódigo los algoritmos deberán presentar el siguiente aspecto:

```
algoritmo  <nombre_algoritmo>
const
   <nombre_de_constante1> = valor1
   ...........................
var
   <tipo_de_dato1>: <nombre_de_variable1> [, <nombre_de_variable2>, .....]
```

```
   ............................
   //Los datos han de ser declarados antes de poder ser utilizados
inicio
   <acción1>
   <acción2>
   //Se utilizará siempre la sangría en las estructuras selectivas y
   //repetitivas.
   .........
   <acciónN>
fin
```

3.4. Ejercicios resueltos

3.1. Se desea calcular independientemente la suma de los números pares en impares comprendidos entre 1 y 200.

Análisis del problema

El algoritmo no necesitaría ninguna variable de entrada, ya que no se le proporciona ningún valor. Como dato de salida se tendrán dos variables (sumapar y sumaimpar) que contendrían los dos valores pedidos. Se necesitará también una variable auxiliar (contador) que irá tomando valores entre 1 y 200.

Después de inicializar el contador y los acumuladores, comienza un bucle que se ejecuta 200 veces. En ese bucle se controla si el contador es par o impar, comprobando si es divisible por dos con el operador **mod**, e incrementando uno u otro acumulador.

Diseño del algoritmo

```
algoritmo ej_3_1
var
   entero : contador,sumapar,sumaimpar
inicio
   contador ← 0
   sumapar ← 0
   sumaimpar ← 0
   repetir
      contador ← contador + 1
      si contador mod 2 = 0 entonces
         sumapar ← sumapar + contador
      si_no
         sumaimpar ← sumaimpar + contador
      fin_si
```

```
   hasta_que contador = 200
   escribir  (sumapar,sumaimpar)
fin
```

3.2. Leer una serie de números enteros positivos distintos de 0 (el último número de la serie debe ser el -99) obtener el número mayor.

Análisis del problema

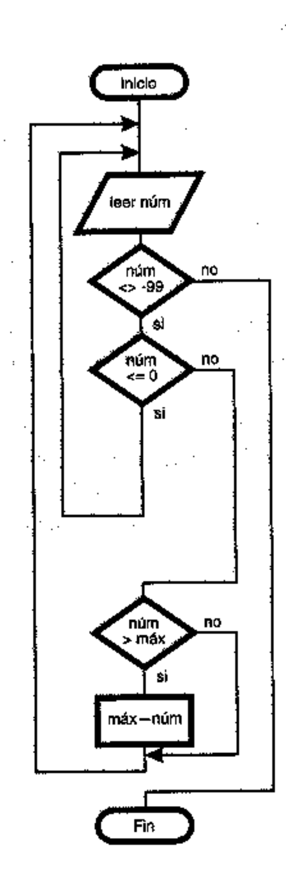
DATOS DE SALIDA: máx (el número mayor de la serie)
DATOS DE ENTRADA: núm (cada uno de los números que introducimos)

Después de leer un número e inicializar máx a ese número, se ejecutará un bucle mientras el último número leído sea distinto de -99. Dentro del bucle se debe controlar que el número sea distinto de 0. Si el número es 0 se vuelve a leer hasta que la condición sea falsa. También hay que comprobar si el último número leído sea mayor que el máximo, en cuyo caso el nuevo máximo será el propio número.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : máx,núm
```



3.3. *Calcular y visualizar la suma y el producto de los números pares comprendidos entre 20 y 400, ambos inclusive.*

Análisis del problema

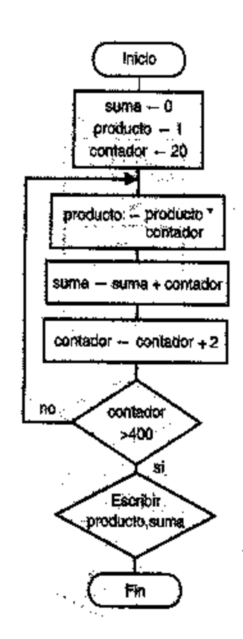
DATOS DE SALIDA: `suma, producto`
DATOS AUXILIARES: `contador`

Para solucionar el algoritmo, se deben inicializar los acumuladores `suma` y `producto` a 0 y la variable `contador` a 20 (puesto que se desea empezar desde 20) e implementar un bucle que se ejecute hasta que la variable `contador` valga 200. Dentro del bucle se irán incrementando los acumuladores `suma` y `producto` con las siguientes expresiones: `suma ← suma+contador` y `producto ← producto*contador`. Una vez realizadas las operaciones se debe incrementar el contador. Como se desea utilizar sólo los números pares, se usará una expresión como `contador ← contador+2`.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero  :  contador,suma,producto
```



3.4. Leer 500 números enteros y obtener cuántos son positivos.

Análisis del problema

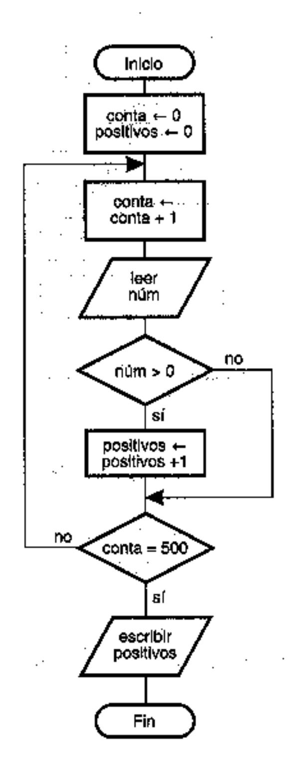
DATOS DE SALIDA: `positivos` (contiene la cantidad de números positivos introducidos)
DATOS DE ENTRADA: `núm` (los números que introducimos)
DATOS AUXILIARES: `conta` (se encarga de contar la cantidad de números introducidos)

Se ha de hacer un bucle que se ejecute 500 veces, controlado por la variable `conta`. Dentro del bucle se lee el número y comprobaremos si es mayor que 0, en cuyo caso se incrementa el contador de positivos (`positivos`).

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : positivos, núm, conta
```



3.5. *Se trata de escribir el algoritmo que permita emitir la factura correspondiente a una compra de un artículo determinado del que se adquieren una o varias unidades. El IVA a aplicar es del 12% y si el precio bruto (precio de venta + IVA) es mayor de 50.000 pesetas, se aplicará un descuento del 5%.*

Análisis del problema

DATOS DE SALIDA:	`bruto` (el precio bruto de la compra, con o sin descuento)
DATOS DE ENTRADA:	`precio` (precio sin IVA), `unidades`
DATOS AUXILIARES:	`neto` (precio sin IVA), `iva` (12% del neto)

Después de leer el precio del artículo y las unidades compradas, se calcula el precio neto (`precio *`

unidades). Se calcula también el IVA (neto * 0.12) y el bruto (neto + iva). Si el bruto es mayor que 50.000 pesetas, se le descuenta un 5% (bruto ← bruto * 0.95).

Al multiplicar el neto por un valor real (0.12) para obtener el IVA, y calcular el bruto a partir del IVA, ambos deben ser datos reales.

Diseño del algoritmo

```
algoritmo  ej_3_5
var
  entero  : precio,neto,unidades
  real : iva,bruto
inicio
  leer  (precio,unidades)
  neto ← precio * unidades
  iva ← neto * 0.12
  bruto ← neto + iva
  si bruto < 50000 entonces
    bruto ← bruto * 0.95
  fin_si
  escribir  (bruto)
fin
```

3.6. Calcular la suma de los cuadrados de los 100 primeros números naturales.

Análisis del problema

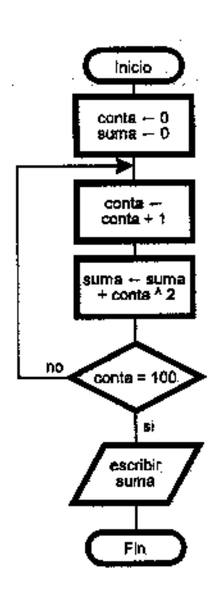
DATOS DE SALIDA: suma (acumula los cuadrados del número)
DATOS AUXILIARES: conta (contador que controla las iteraciones del bucle)

Para realizar este programa es necesario un bucle que se repita 100 veces y que se controlará por la variable conta. Dentro de dicho bucle se incrementará el contador y se acumulará el cuadrado del contador en la variable suma.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : suma, conta
```



3.7. Sumar los números pares del 2 al 100 e imprimir su valor.

Análisis del problema

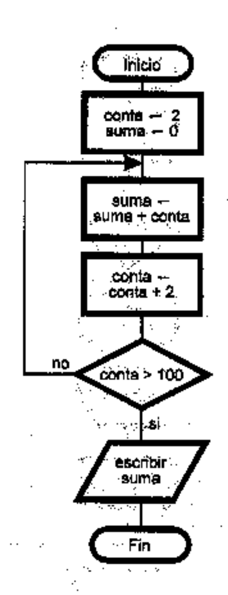
DATOS DE SALIDA: `suma` (contiene la suma de los números pares)
DATOS AUXILIARES: `conta` (contador que va sacando los números pares)

Se trata de hacer un bucle en el que un contador (`conta`) vaya incrementando su valor de dos en dos y, mediante el acumulador `suma`, y acumulando los sucesivos valores de dicho contador. El contador se deberá inicializar a 2 para que se saquen números pares. El bucle se repetirá hasta que `conta` sea mayor que 100.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : conta,suma
```



3.8. Sumar 10 números introducidos por teclado.

Análisis del problema

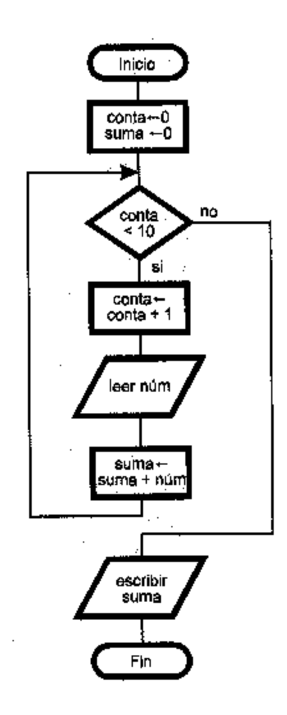
DATOS DE SALIDA: `suma` (suma de los números)
DATOS DE ENTRADA: `núm` (los números que introducimos por teclado)
DATOS AUXILIARES: `conta` (contador que controla la cantidad de números introducidos)

Después de inicializar `conta` y `suma` a 0, se debe implementar un bucle que se ejecute 10 veces. En dicho bucle se incrementará el contador `conta` en una unidad, se introducirá por teclado `núm` y se acumulará su valor en `suma`. El bucle se ejecutará mientras que `conta` sea menor o igual que 10.

Diseño del algoritmo

TABLA DE VARIABLES

```
entero : suma, núm, conta
```



3.9. Calcular la media de 50 números introducidos por teclado y visualizar su resultado.

Análisis del problema

DATOS DE SALIDA: `media` (contiene la media de los cincuenta números)
DATOS DE ENTRADA: `núm` (cada uno de los cincuenta números introducidos por teclado)
DATOS AUXILIARES: `conta` (contador que controla la cantidad de números introducidos), `suma` (acumula el valor de los números)

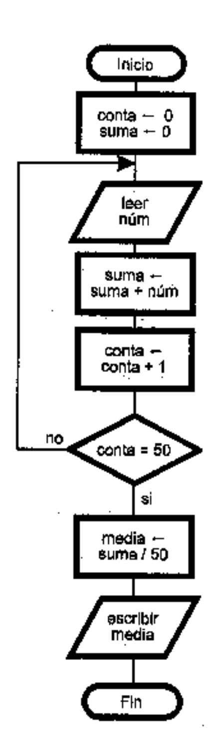
Después de inicializar `conta` y `suma`, se realiza un bucle que se repetirá 50 veces. En dicho bucle se lee un número (`núm`), se acumula su valor en `suma` y se incrementa el contador `conta`. El bucle se repetirá hasta que `conta` sea igual a 50. Una vez fuera del bucle se calcula la media (`suma / 50`) y se escribe el resultado.

Diseño del algoritmo

TABLA DE VARIABLES

```
entero : núm,conta,suma
```

```
real : media
```



3.10. *Visualizar los múltiplos de 4 comprendidos entre 4 y N, donde N es un número introducido por teclado.*

Análisis del problema

DATOS DE SALIDA:	`múltiplo` (cada uno de los múltiplos de 4)
DATOS DE ENTRADA:	`N` (número qué indica hasta que múltiplo vamos a visualizar)
DATOS AUXILIARES:	`conta` (contador que servirá para calcular los múltiplos de 4)

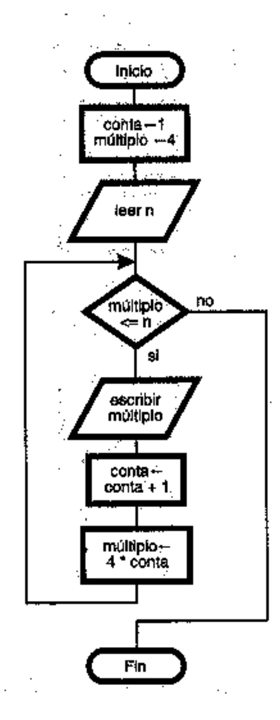
Para obtener los múltiplos de cuatro se pueden utilizar varios métodos. Un método consiste en sacar números correlativos entre 4 y `N` y para cada uno comprobar si es múltiplo de 4 mediante el operador **mod**. Otro método sería utilizar un contador que arrancando en 4 se fuera incrementado de 4 en 4. Aquí simplemente se irá multiplicando la constante 4 por el contador, que tomará valores a partir de 1.

Para realizar el algoritmo, después de inicializar `conta` a 1 y `múltiplo` a 4 y leer el número de múltiplos que se desean visualizar, se debe ejecutar un bucle mientras que `múltiplo` sea menor que `N`. Dentro del bucle hay que visualizar `múltiplo`, incrementar el contador en 1 y calcular el nuevo múltiplo (`4 * conta`).

Diseño del algoritmo

TABLA DE VARIABLES

entero : `múltiplo,N,conta`



3.11. *Realizar un diagrama que permita realizar un contador e imprimir los 100 primeros números enteros.*

Análisis del problema

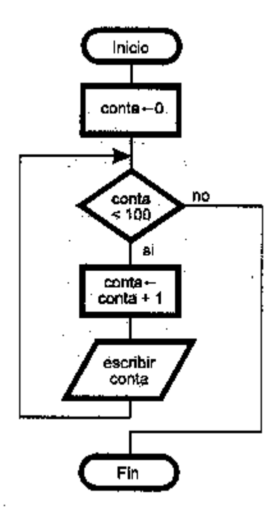
DATOS DE SALIDA:	Los números enteros entre 1 y 100
DATOS AUXILIARES:	`conta` (controla el número de veces que se ejecuta el bucle)

Se debe ejecutar un bucle 100 veces. Dentro del bucle se incrementa en 1 el contador y se visualiza éste. El bucle se ejecutará mientras `conta` sea menor que 100. Previamente a la realización del bucle, `conta` se inicializará a 0.

Diseño del algoritmo

TABLA DE VARIABLES

entero : conta



3.12. Dados 10 números enteros que introduciremos por teclado, visualizar la suma de los números pares de la lista, cuántos números pares existen y cuál es la media aritmética de los números impares.

Diseño del algoritmo

DATOS DE SALIDA:	spar (suma de pares), npar (cantidad de números pares), media (media de números impares)
DATOS DE ENTRADA:	num (cada uno de los números introducidos por teclado)
DATOS AUXILIARES:	conta (contador que controla la cantidad de números introducidos), simpar (suma de los números impares), nimpar (cantidad de números impares)

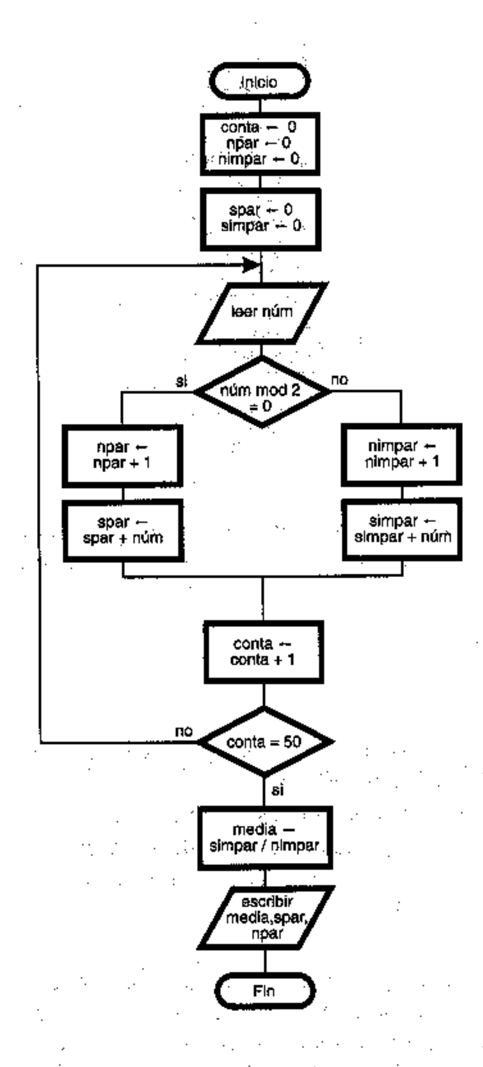
Se ha de realizar un bucle 10 veces. En ese bucle se introduce un número y se comprueba si es par mediante el operador **mod**. Si es par se incrementa el contador de pares y se acumula su valor en el acumulador de pares. En caso contrario se realizan las mismas acciones con el contador y el acumulador de impares. Dentro del bucle también se ha de incrementar el contador conta en una unidad. El bucle se realizará hasta que conta sea igual a 10.

Ya fuera del bucle se calcula la media de impares (simpar / nimpar) y se escribe spar, npar y media.

Diseño del algoritmo

TABLA DE VARIABLES

```
entero  :  conta,núm,spar,npar,simpar,nimpar
real  : media
```



3.13. *Calcular la nota media por alumno de una clase de a alumnos. Cada alumno podrá tener un número n de notas distinto.*

Análisis del problema

DATOS DE SALIDA: `media` (media de cada alumno que también utilizaremos para acumular las notas)
DATOS DE ENTRADA: `a` (número de alumnos), `n` (número de notas de cada alumno), `nota` (nota de cada alumno en cada una de las asignaturas)
DATOS AUXILIARES: `contaa` (contador de alumnos), `contan` (contador de notas)

Para realizar este algoritmo se han de realizar dos bucles anidados. El primero se repetirá tantas veces como alumnos. El bucle interno se ejecutará por cada alumno tantas veces como notas tenga éste.

En el bucle de los alumnos se lee el número de notas e inicializar las variables `media` y `contan` a 0. Aquí comenzará el bucle de las notas en el que leeremos una nota, se acumula en la variable `media` y se incrementa el contador de notas. Este bucle se ejecutará hasta que el contador de notas sea igual a `n` (número de notas).

Finalizado el bucle interno, pero todavía en el bucle de los alumnos, se calcula la media (`media / n`), se escribe y se incrementa el contador de alumnos. El programa se ejecutará hasta que `contaa` sea igual a `a`.

Diseño del algoritmo

TABLA DE VARIABLES

```
entero  : a,n,contaa,contan
real  : media,nota
```

inicio

contaa ← 0

leer a

leer n

contan ← 0
media ← 0

leer nota

media ← media + nota

contan ← contan + 1

contan = n (no / si)

media ← media / n

escribir media

contaa ← contaa + 1

contaa = a (no / si)

Fin

4

Introducción a la programación estructurada

4.1. Programación estructurada

La programación estructurada es un conjunto de técnicas para desarrollar algoritmos fáciles de escribir, verificar, leer y modificar. La programación estructurada utiliza:

- **diseño descendente**. Consiste en diseñar los algoritmos en etapas, yendo de los conceptos generales a los de detalle. El diseño descendente se verá completado y ampliado con el modular.
- **recursos abstractos**. En cada descomposición de una acción compleja se supone que todas las partes resultantes están ya resueltas, posponiendo su realización para el siguiente refinamiento.
- **estructuras básicas**. Los algoritmos deberán ser escritos utilizando únicamente tres tipos de estructuras básicas.

4.2. Teorema de Böhm y Jacopini

Para que la programación sea estructurada, los programas han de ser propios. Un programa se define como propio si cumple las siguientes características:

- tiene un solo punto de entrada y uno de salida.
- toda acción del algoritmo es accesible, es decir, existe al menos un camino que va desde el inicio hasta el fin del algoritmo, se puede seguir y pasa a través de dicha acción.
- no posee lazos o bucles infinitos.

El teorema de Böhm y Jacopini dice que: *«un programa propio puede ser escrito utilizando*

únicamente tres tipos de estructuras: secuencial, selectiva y repetitiva».

De este teorema se deduce que se han de diseñar los algoritmos empleando exclusivamente dichas estructuras, la cuales, como tienen un único punto de entrada y un único punto de salida, harán que nuestros programas sean propios.

4.3. Estructuras de control

A las estructuras secuencial, selectiva y repetitiva se las denomina estructuras de control debido a que controlan el modo de ejecución del programa.

4.3.1. Estructuras secuenciales

Se caracterizan porque una acción se ejecuta detrás de otra. El flujo del programa coincide con el orden físico en el que se han ido poniendo las instrucciones.

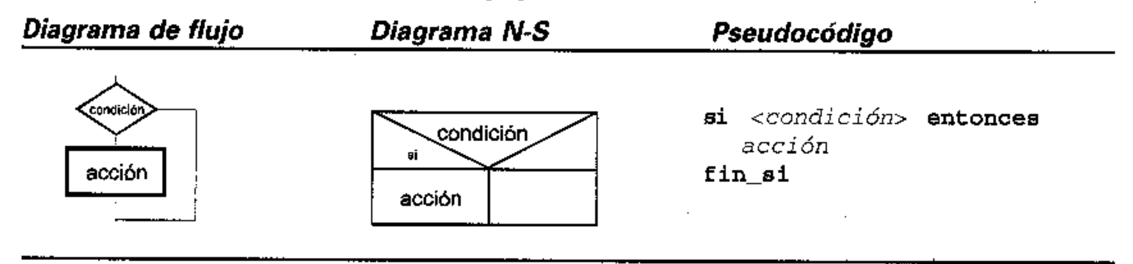
Diagrama de flujo	***Diagrama N-S***	***Pseudocódigo***
acción 1 acción 2 acción n	acción 1 acción 2 acción n	`acción 1` `acción 2` `acción n`

4.3.2. Estructuras selectivas

Se ejecutan unas acciones u otras según se cumpla o no una determinada condición; pueden ser *simples*, *dobles* o *múltiples*.

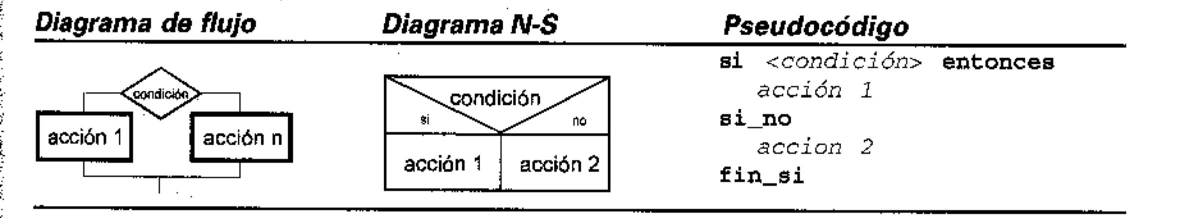
simples

Se evalúa la condición y si ésta da como resultado verdad se ejecuta una determinada acción o grupo de acciones; en caso contrario se saltan dicho grupo de acciones.



dobles

Cuando el resultado de evaluar la condición es verdad se ejecutará una determinada acción o grupo de acciones y si el resultado es falso otra acción o grupo de acciones diferentes



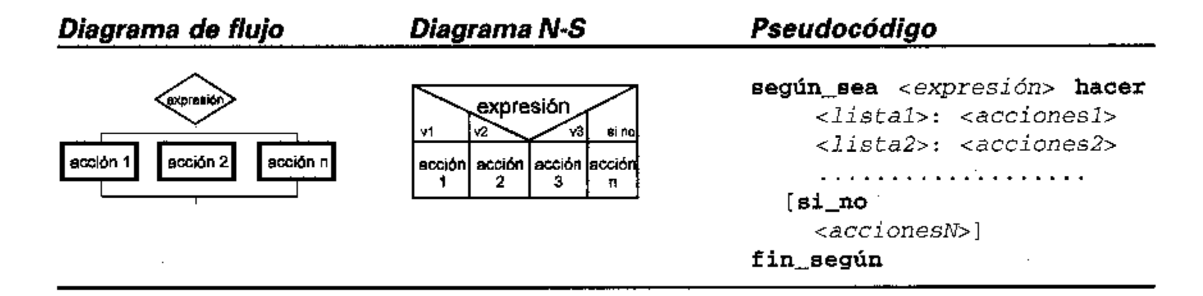
múltiple

Se ejecutarán unas acciones u otras según el resultado que se obtenga al evaluar una expresión. Se considera que dicho resultado ha de ser de un tipo ordinal, es decir de un tipo de datos en el que cada uno de los elementos que constituyen el tipo, excepto el primero y el último, tiene un único predecesor y un único sucesor.

Cada grupo de acciones se encontrará ligado con: un valor, varios valores separados por comas, un rango, expresado como valor_inicial..valor_final o una mezcla de valores y rangos.

Se ejecutarán únicamente las acciones del primer grupo que, entre los valores a los que está ligado, cuente con el obtenido al evaluar la expresión. Cuando el valor obtenido al evaluar la expresión no esté presente en ninguna lista de valores se ejecutarían las acciones establecidas en la cláusula **si_no**, si existiese dicha cláusula.

Diagrama de flujo **Diagrama N-S** **Pseudocódigo**




```
según_sea <expresión> hacer
    <lista1>: <acciones1>
    <lista2>: <acciones2>
    ...................
  [si_no
    <accionesN>]
fin_según
```

4.3.3. *Estructuras repetitivas*

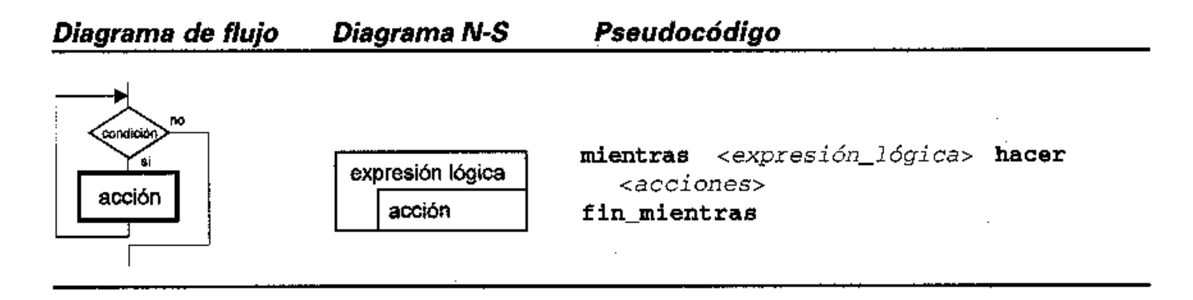
Las acciones del cuerpo del bucle se repiten mientras o hasta que se cumpla una determinada condición. Es frecuente el uso de contadores o banderas para controlar un bucle. También se utilizan con esta finalidad los centinelas.

Un centinela es un valor anómalo dado a una variable que permite detectar cuándo se desea terminar de repetir las acciones que constituyen el cuerpo del bucle. Por ejemplo, se puede diseñar un bucle que pida el nombre y la nota de una serie de alumnos y establecer que termine cuando se le introduzca un '*' como nombre. Podemos considerar tres tipos básicos de estructuras repetitivas: **mientras**, **hasta**, **desde**.

mientras

Lo que caracteriza este tipo de estructura es que las acciones del cuerpo del bucle se realizan cuando la condición es cierta. Además, se pregunta por la condición al principio, de donde se deduce que dichas acciones se podrán ejecutar de 0 a N veces.

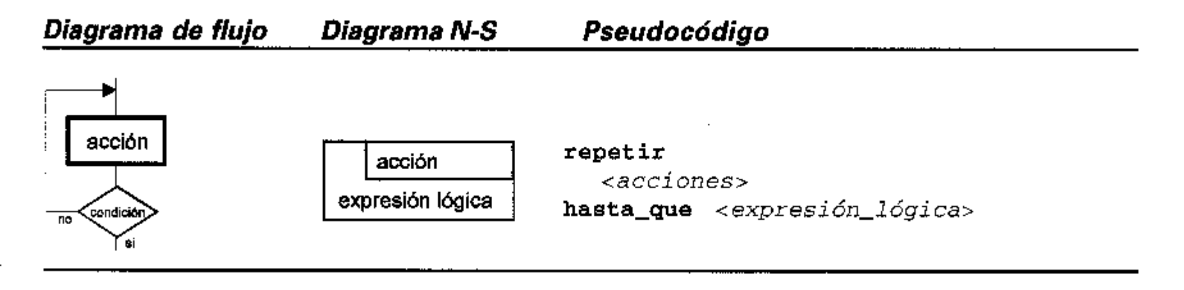
Diagrama de flujo **Diagrama N-S** **Pseudocódigo**




```
mientras <expresión_lógica> hacer
    <acciones>
fin_mientras
```

hasta

Las acciones del interior del bucle se ejecutan una vez y continúan repitiéndose mientras que la condición sea falsa. Se interroga por la condición al final del bucle.




```
repetir
   <acciones>
hasta_que <expresión_lógica>
```

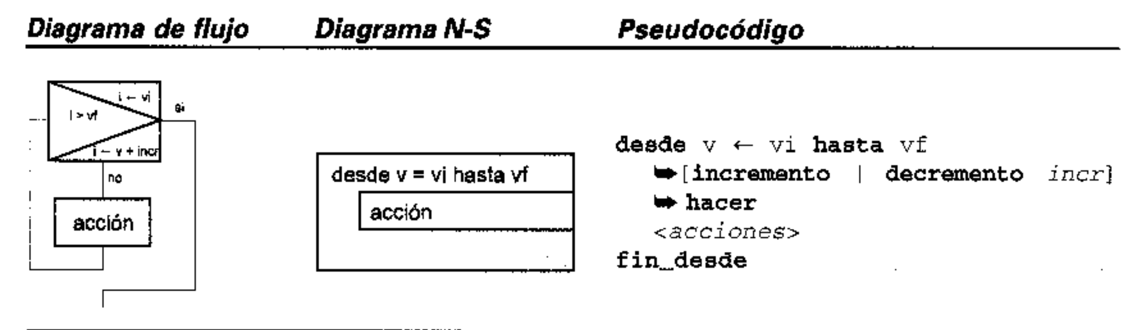
desde

Se utiliza cuando se conoce, con anterioridad a que empiece a ejecutarse el bucle, el número de veces que se va a iterar.

La estructura **desde** comienza con un valor inicial de la variable índice y las acciones especificadas se ejecutan a menos que el valor inicial sea mayor que el valor final. La variable índice se incrementa en 1, o en el valor que especifiquemos, y si este nuevo valor no excede al final se ejecutan de nuevo las acciones.

Si establecemos que la variable índice se decremente en cada iteración el valor inicial deberá ser superior al final. Consideramos siempre la variable índice de tipo entero.

Es posible sustituir una estructura **desde** por otra de tipo **mientras** controlada por un contador.




```
desde v ← vi hasta vf
   ➡[incremento | decremento incr]
   ➡ hacer
   <acciones>
fin_desde
```

4.3.4. Estructuras anidadas

Tanto las estructuras selectivas como las repetitivas pueden ser anidadas, e introducidas unas en el interior de otras.

La estructura selectiva múltiple es un caso especial de varias estructuras selectivas dobles anidadas en la rama **si_no**.

```
si <condición1> entonces
   <acciones1>
si_no
   si <condición2> entonces
      <acciones2>
```

```
   si_no
      si <condición3> entonces
         <acciones3>
      si_no
         <accionesX>
      fin_si
   fin_si
fin_si
```

Cuando se inserta un bucle dentro de otro la estructura interna ha de estar totalmente incluida dentro de la externa. Es posible anidar cualquier tipo de estructura repetitiva. Si se anidan dos estructuras **desde**, para cada valor de la variable índice del ciclo externo se debe ejecutar totalmente el bucle interno.

```
desde v1 ← vi1 hasta vf1 hacer
   desde v2 ← vi2 hasta vf2 hacer
      <acciones>
   fin_desde
fin_desde
```

Las acciones que componen el cuerpo del bucle más interno se ejecutarán el siguiente número de veces:

```
(vf1 - vi1 + 1) * (vf2 - vi2 + 1)
```

4.4. Ejercicios resueltos

4.1. Dados tres números, deducir cuál es el central

Análisis del problema

DATOS DE SALIDA: `Central` (el número central)
DATOS DE ENTRADA: `A`, `B` y `C` (los números que vamos a comparar)

Se trata de ir comparando los tres números entre sí, utilizando selecciones de una sola comparación anidadas entre sí, ya que así se ahorran comparaciones (sólo utilizamos 5). Si se utilizan comparaciones con operadores lógicos del tipo `A<B` **and** `B>C`, se necesitarían seis estructuras selectivas por lo que se estarían haciendo dos comparaciones por selección.

Diseño del algoritmo (con comparaciones dobles)

```
algoritmo Ejercicio_4_1
var
```

```
   entero : a, b, c, central
inicio
   leer (a, b, c)
   si (a < b) y (b < c) entonces
      central ← b
   fin_si
   si (a < c) y (c < b) entonces
      central ← c
   fin_si
   si (b < a) y (a < c) entonces
      central ← a
   fin_si
   si (b < c) y (c < a) entonces
      central ← c
   fin_si
   si (c < a) y (a < b) entonces
      central ← a
   fin_si
   si (c < b) y (b < a) entonces
      central ← b
   fin_si
   escribir (central)
fin
```

Diseño del algoritmo (con comparaciones simples)

```
algoritmo ejercicio_4_1
var
   entero : a, b, c, central
inicio
   leer (a, b, c)
   si a > b entonces
      si b > c entonces
         central ← b
      si_no
         si a > c entonces
            central ← c
         si_no
            central ← a
         fin_si
    si_no
      si a > c entonces
         central ← a
      si_no
         si c > b entonces
            central ← b
         si_no
            central ← a
```

```
        fin_si
      fin_si
   fin_si
       escribir  (central)
fin
```

4.2. Calcular la raíz cuadrada de un número y escribir su resultado.

Análisis del problema

Como dato de salida se tendrá la raíz y como entrada el número. Lo único que hay que hacer es asignar a `raíz` la raíz cuadrada del número, siempre que éste no sea negativo, ya que en ese caso no tendría una solución real. Se utilizará la función **raízcua** si se considera implementada; en caso contrario, deberá utilizarse la exponenciación.

Diseño del algoritmo

```
algoritmo ejercicio_4_2
var
   entero : n
   real  : raíz
inicio
   leer (n)
   si n < 0 entonces
      escribir ('no hay solución real')
   si_no
      raíz ← n ^ (1/2)
      escribir (raíz)
   fin_si
fin
```

4.3. Escribir los diferentes métodos para deducir si una variable o expresión numérica es par.

La forma de deducción se hace generalmente comprobando de alguna forma que el número o expresión numérica es divisible por dos. La forma más común sería utilizando el operador **mod**, el resto de la división entera. La variable a comprobar, `var`, será par si se cumple la condición

`var mod 2 = 0`

y en caso de no disponer del operador **mod** y sabiendo que el resto es

`resto = dividendo - divisor * ent(dividendo / divisor)`

se trata de ver si `var` es par si se cumple la expresión siguiente

`var - 2 * ent(var / 2) = 0`

Otra variante podría ser comprobar si la división real de `var` entre 2 es igual a la división entera de `var` entre 2

```
var / 2 = var div 2
```

o, siguiendo el mismo método si

```
var / 2 = ent(var / 2)
```

En algunos casos, estos ejemplos pueden llevar a error según sea la precisión que la computadora obtenga en el cálculo de la expresión.

4.4. *Determinar el precio de un billete de ida y vuelta en ferrocarril, conociendo la distancia a recorrer y sabiendo que si el número de días de estancia es superior a siete y la distancia superior a 800 kilómetros el billete tiene una reducción del 30%. El precio por kilómetro es de 2,5 pesetas.*

Análisis del problema

DATOS DE SALIDA: Precio del billete
DATOS DE ENTRADA: distancia a recorrer, días de estancia

Se lee la distancia y el número de días y se halla el precio del billete de ida y vuelta (`precio = distancia * 2 * 2.5`). Se comprueba si la distancia es superior a 800 Km. y los días de estancia a 7 y si es cierto se aplica una reducción del 30%.

Diseño del algoritmo

```
algoritmo ejercicio_4_4
var
  entero : distancia,días
  real : precio
inicio
  leer (distancia,días)
  precio ← distancia * 2 * 2.5
  si días > 7 y distancia > 800 entonces
      precio ← precio * 0.3
  fin_si
  escribir (precio)
fin
```

4.5. *Diseñar un algoritmo en el que a partir de una fecha introducida por teclado con el formato DÍA, MES, AÑO, se obtenga la fecha del día siguiente.*

Análisis del problema

DATOS DE SALIDA: `dds`, `mms`, `aas` (día, mes y año del día siguiente)
DATOS DE ENTRADA: `dd`, `mm`, `aa` (día mes y año del día actual)

En principio lo único que habría que hacer es sumar una unidad al día. Si el día actual es menor que 28 (número de días del mes que menos días tiene) no sucede nada, pero se debe comprobar si al sumar un día ha habido cambio de mes o de año, para lo que se comprueba los días que tiene el mes, teniendo también en cuenta los años bisiestos. También se debe comprobar si es el último día del año en cuyo caso se incrementa también el año. Se supone que la fecha introducida es correcta.

Diseño del algoritmo

```
algoritmo ejercicio_4_5
var
  entero : dd, mm, aa, dds, mms, aas
inicio
  leer(dd,mm,aa)
  dds ← dd + 1
  mms ← mm
  aas ← aa
  si dd >= 28 entonces
    según_sea mm hacer
      //si el mes tiene treinta días
      4,6,9,11  : si dds > 30 entonces
                    dds ← 1
                    mms ← mm + 1
                  fin_si
      //si el mes es febrero
      2 :  si (dds > 29) o (aa mod 4 <> 0) entonces
              dds ← 1
              mms ← 3
           fin_si
      //si el mes tiene 31 días y no es diciembre
      1,3,5,7,8,10  : si dds > 31 entonces
                         dds ← 1
                         mms ← mm + 1
                      fin_si
    si_no
      //si el mes es diciembre
      si dds > 31 entonces
        dds ← 1
        mms ← 1
        aas ← aa + 1
      fin_si
    fin_según
  fin_si
  escribir(dds,mms,aas)
fin
```

4.6. *Se desea realizar una estadística de los pesos de los alumnos de un colegio de acuerdo a la siguiente tabla:*

Alumnos de menos de 40 kg.
Alumnos entre 40 y 50 kg.
Alumnos de más de 50 y menos de 60 kg.
Alumnos de más o igual a 60 kg.

La entrada de los pesos de los alumnos se terminará cuando se introduzca el valor centinela -99. Al final se desea obtener cuántos alumnos hay en cada uno de los baremos.

Análisis del problema

DATOS DE SALIDA: `conta1, conta2, conta3, conta4` (contadores de cada uno de los baremos)
DATOS DE ENTRADA: `peso` (peso de cada uno de los alumnos)

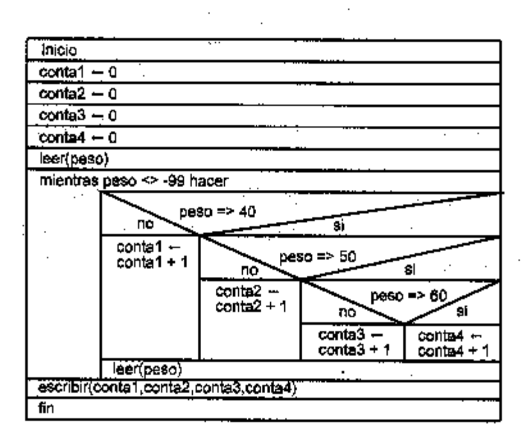
Se construye un bucle que se ejecute hasta que el peso introducido sea igual a -99. Dentro del bucle se comprueba con una serie de estructuras **si** anidadas a qué lugar de la tabla corresponde el peso, incrementándose los contadores correspondientes. Cuando se haya salido del bucle se escribirán los contadores.

Nótese que ha de leer el peso una vez antes de entrar en el bucle y otra vez como última instrucción de éste, pues de lo contrario se incluirá el centinela (-99) en la estadística la última vez que se ejecute el bucle.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : conta1, conta2, conta3, conta4
real : peso
```



4.7. Realizar un algoritmo que averigüe si dados dos números introducidos por teclado, uno es divisor del otro.

Análisis del problema

DATOS DE SALIDA:	El mensaje que nos dice si es o no divisor
DATOS DE ENTRADA:	`núm1, núm2` (los números que introducimos por teclado)
DATOS AUXILIARES:	`divisor` (variable lógica que será cierta si el segundo número es divisor del primero)

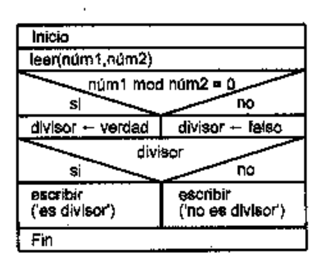
En este problema se utiliza el operador **mod** para comprobar si `núm2` es divisor de `núm1` Si el resto de dividir los dos números es 0 se establece la variable `divisor` como cierta, y en caso contrario como falsa.

Preguntando por el valor del divisor se obtiene un mensaje que nos indicará si `núm2` es o no divisor de `núm2`

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : núm1, núm2
lógico : divisor
```



4.8. Analizar los distintos métodos de realizar la suma de T números introducidos por teclado.

Para realizar la suma de T números será imprescindible realizar un bucle que se ejecute T veces y que incluya una operación de lectura y un acumulador. Al conocer de antemano el número de veces que se ejecuta el bucle (T), lo más adecuado será utilizar un bucle de tipo **desde**:

```
.
.
desde i ← 1 hasta T hacer
```

```
    leer(número)
    suma ← suma + número
fin_desde
.
.
```

El mismo bucle se puede realizar con una estructura **mientras** o **repetir**, en cuyo caso dentro del bucle se incluirá también un contador.

Con una estructura mientras:

```
.
.
conta ← 0
mientras conta < T hacer
    leer(número)
    suma ← suma + número
    conta ← conta + 1
fin_mientras
.
.
```

Con una estructura repetir:

```
.
.
conta ← 0
repetir
    leer(número)
    suma ← suma + número
    conta ← conta + 1
hasta_que conta = T
.
.
```

4.9. Se desea un algoritmo que realice la operación de suma o resta de dos números leídos del teclado en función de la respuesta S o R (suma o resta) que se dé a un mensaje de petición de datos.

Análisis del problema

DATOS DE SALIDA:	`resultado` (resultado de la operación suma o resta de los dos números)
DATOS DE ENTRADA:	`a, b` (los números a operar), `operación` (tipo de la operación a efectuar)

Después de leer los números `a` y `b`, un mensaje ha de solicitar que pulsemos una tecla S o una tecla R para realizar una u otra operación. Aquí se incluye un bucle que se repita hasta que la entrada sea `S` o `R`, para evitar errores de entrada. En función de la respuesta se calcula el resultado que será `a + b`, en caso que la respuesta sea `S`, o `a - b` en caso que la respuesta sea `R`.

Diseño del algoritmo

```
algoritmo ejercicio_4_9
var
   entero : a,b,resultado
   carácter : operación
inicio
   leer(a,b)
   escribir('Pulse S para sumar, R para restar')
   repetir
      leer(operación)
   hasta_que (operación = 'S') o (operación = 'R')
   si operación = 'S' entonces
      resultado ← a + b
   si_no
      resultado ← a - b
   fin_si
   escribir(resultado)
fin
```

4.10. *Escribir un algoritmo que lea un número y deduzca si está entre 10 y 100, ambos inclusive.*

Análisis del problema

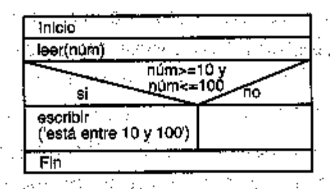
DATOS DE SALIDA:	El mensaje que indica si el número está entre esos límites
DATOS DE ENTRADA:	núm (el número que se desea comprobar)

Mediante una comparación múltiple se comprueba si el número está entre esos límites, en cuyo caso aparecerá un mensaje.

Diseño del algoritmo

TABLA DE VARIABLES

```
entero : núm
```



4.11. *Se dispone de las calificaciones de los alumnos de un curso de informática correspondientes a*

las asignaturas de BASIC, FORTRAN y PASCAL. Diseñar un algoritmo que calcule la media de cada alumno.

Análisis del problema

DATOS DE SALIDA: `media` (una por alumno)
DATOS DE ENTRADA: `nota` (tres por alumno) y `n` (número de alumnos)
DATOS AUXILIARES: `i`, `j` (variables que controlan los bucles)

Hay que leer el número de alumnos para conocer las iteraciones que debe ejecutarse el bucle (si se utilizara un bucle **repetir** o **mientras** se debería establecer alguna otra condición de salida). Dentro del bucle principal se anida otro bucle que se repite tres veces por alumno (cada alumno tiene tres notas) en el que leemos la nota y la acumulamos en la variable `media`. Al finalizar el bucle interno se halla y se escribe la media.

Diseño del algoritmo

```
algoritmo ejercicio_4_11
var
   entero : n,i,j
   real : media, nota
inicio
   leer(n) //número de alumnos
   desde i = 1 hasta n hacer
      media ← 0
      desde j = 1 hasta 3 hacer
         leer(nota)
         media ← media + nota
      fin_desde
      media ← media/3
      escribir(media)
   fin_desde
fin
```

4.12. *Escribir el ordinograma y el pseudocódigo que calcule la suma de los 50 primeros números enteros.*

Análisis del problema

DATOS DE SALIDA: `suma` (suma de los primeros 50 números enteros)
DATOS AUXILIARES: `conta` (contador que irá sacando los números enteros)

Se trata de hacer un bucle en el que se incremente un contador y que se acumule el valor de éste en un

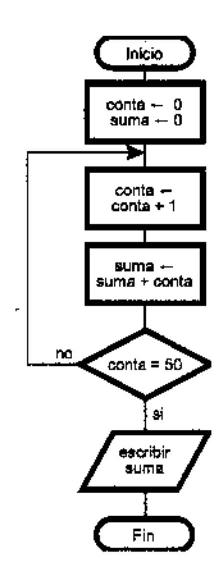
acumulador (suma). El bucle se ejecutará hasta que el contador sea igual a 50.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : suma,conta
```

Ordinograma



Pseudocódigo

```
algoritmo  ejercicio_4_12
var
   entero :  suma,conta
inicio
   suma ← 0
   conta ← 0
   repetir
      conta ← conta + 1
      suma ← suma + conta
   hasta_que conta = 50
   escribir(suma)
fin
```

4.13. Calcular y escribir los cuadrados de una serie de números distintos de 0 leídos desde el teclado

Análisis del problema

DATOS DE SALIDA: `cuadrado`
DATOS DE ENTRADA: `núm`

Dado que se desconoce la cantidad de números leidos, se debe utilizar un bucle **`repetir`** o **`mientras`**. Se construye un bucle hasta que el número sea 0. Dentro del bucle se lee `núm` y se calcula su cuadrado. Para no incluir el último número -el cero-, se ha de leer antes de entrar al bucle y otra vez al final de éste.

Diseño del algoritmo

```
algoritmo ejercicio_4_13
var
  entero : núm, cuadrado
inicio
  leer(núm)
  mientras núm <> 0 hacer
    cuadrado ← núm * núm
    escribir(cuadrado)
    leer(núm)
  fin_mientras
fin
```

4.14. Un capital C está situado a un tipo de interés R ¿Se doblará el capital al término de dos años?

Análisis del problema

DATOS DE SALIDA: El mensaje que nos dice si el capital se dobla o no
DATOS DE ENTRADA: `C` (capital), `R` (interés)
DATOS AUXILIARES: `CF` (capital al final del periodo)

Después de introducir el capital y el interés por teclado, se calcula el capital que se producirá en dos años por la fórmula del interés compuesto:

$$CF = C(1+R)^2$$

Si `CF` es igual a `C*2` aparecerá el mensaje diciendo que el capital se doblará.

Diseño del algoritmo

```
algoritmo Ejercicio_4_14
var
   entero : C
   real : R,CF
inicio
   leer(C,R)
   CF ← C*(1+R/100)^2
   si C*2 <= CF entonces
      escribir('El capital se doblará o superará el doble')
   si_no
      escribir('El capital no se doblará')
   fin_si
fin
```

4.15. Leer una serie de números desde el terminal y calcular su media. La marca de fin de lectura será el número -999.

Análisis del problema

DATOS DE SALIDA:	media (media de los números)
DATOS DE ENTRADA:	núm (cada uno de los números)
DATOS AUXILIARES:	conta (cuenta los números introducidos excepto el -999), suma (suma los números excepto el -999)

Se debe construir un bucle que se repita hasta que el número introducido sea -999. Como ya se vio más arriba (ejercicio 4.13.) se debe leer antes de comenzar el número y al final del bucle. Dentro del bucle se incrementará un contador, necesario para poder calcular la media, y un acumulador guardará la suma parcial de los números introducidos. Una vez fuera del bucle se calcula e imprime la media.

Diseño del algoritmo

```
algoritmo ejercicio_4_15
var
   entero : conta, suma, núm
   real : media
inicio
   conta ← 0
   suma ← 0
   leer(núm)
   mientras núm <> -999 hacer
      conta ← conta + 1
```

```
      suma ← suma + núm
      leer(núm)
   fin_mientras
   media ← suma / conta
   escribir(media)
fin
```

4.16. Se considera la serie definida por:

$$a_1=0,\ a_2=1,\ a_n=3*a_{n-1}+2*a_{n-2}\ (para\ n >=3)$$

Se desea obtener el valor y el rango del primer término que sea mayor o igual que 1000.

Análisis del problema

DATOS DE SALIDA:	`i` (rango del primer término de la serie mayor o igual que 1000), `último` (valor de dicho término)
DATOS AUXILIARES:	`penúltimo` (penúltimo valor de la serie), `valor` (valor actual)

Se calcula el término por tanteo, es decir a partir del tercer término, se ejecuta un bucle que se realice mientras que el valor del término sea menor que 1000.

Para ello se inicializa `i` a 2, `último` a 1 y `penúltimo` a 0. Dentro del bucle se calcula el valor (`3 * último + 2 * penúltimo`). Antes de finalizar el bucle se deben actualizar los valores de `penúltimo` (`penúltimo ← último`) y `último` (`último ← valor`). También se ha de incrementar el contador `i` en una unidad. Al finalizar el bucle, primer valor mayor o igual a 1000 será `último`, y el término `i`.

Diseño del algoritmo

```
algoritmo ejercicio_4_16
var
   entero : i,valor,último,penúltimo
inicio
   i ← 2
   último ← 1
   penúltimo ← 0
   mientras último < 1000  hacer
      valor ← 3 * último + 2 * penúltimo
      i ← i+1
      penúltimo ← último
      último ← valor
   fin_mientras
   escribir(último,i)
fin
```

4.17. Escribir un algoritmo que permita calcular X^n, donde:

X puede ser cualquier número real distinto de 0

n puede ser cualquier entero positivo, negativo o nulo

Nota: suponemos que no está implementado el operador de exponenciación.

Análisis del problema

DATOS DE SALIDA:	potencia (resultado de elevar X a la n)
DATOS DE ENTRADA:	X (base de la exponenciación), n (exponente)
DATOS AUXILIARES:	conta (contador que controla el número de veces que se multiplica X por sí mismo), solución (será falso si no hay solución)

Después de introducir X y n, se comprueba n. Si es 0, potencia valdrá 1. Si es positivo se ha de multiplicar X n veces, y ello se hará mediante un bucle en el que se vayan guardando en la variable producto las sucesivas multiplicaciones. En el bucle un contador irá controlando las veces que se ejecuta el bucle, que finalizará cuando sea igual a n. Si n es negativo, primero se comprueba que X sea distinto de cero, pues en ese caso no tendrá solución. En caso contrario se procederá de la misma forma que si es positivo, pero la potencia será 1/potencia.

Diseño del algoritmo

```
algoritmo ejercicio_4_17
var
   entero : n,conta
   real : X,potencia
   lógico : solución
inicio
   leer(X,n)
   solución ← verdad
   si n = 0 entonces
      potencia ← 1
   si_no
      si n > 0 entonces
         potencia ← 1
         conta ← 0
         repetir
            potencia ← potencia * X
            conta ← conta + 1
         hasta_que conta = n
      si_no
         si X = 0 entonces
            escribir('No hay solución')
            solución ← falso
         si_no
            potencia ← 1
```

```
            conta ← 0
            repetir
               potencia ← potencia * X
               conta ← conta -1
            hasta_que conta = n
            potencia ← 1 / potencia
         fin_si
      fin_si
   fin_si
   si solución entonces
      escribir(potencia)
   fin_si
fin
```

4.18. Se desea leer desde teclado una serie de números hasta que aparezca alguno menor que 1000.

Análisis del problema

En este algoritmo, sólo hay un dato de entrada, que será cada uno de los números que leemos, y no tenemos ningún dato de salida. No hay más que ejecutar un bucle hasta que el número leído sea menor de 1000.

Diseño del algoritmo

```
algoritmo ejercicio_4_18
var
   real : núm
inicio
repetir
   leer(núm)
hasta_que núm < 1000    //sólo se admiten los menores de 1000
fin
```

4.19. Se desea obtener los cuadrados de todos los números leídos desde un archivo hasta que se encuentre el número 0.

Análisis del problema

Este ejercicio es muy similar al anterior aunque se introduce la noción de archivo que será vista en profundidad más adelante. Por el momento no vamos a tener en cuenta las operaciones de abrir y cerrar el archivo, así como la marca de fin de archivo o la definición de su estructura. Para leer números desde el archivo lo único que vamos a hacer es una operación de lectura indicando que se lee desde un archivo:

```
leer(nombre del archivo,lista de variables)
```

Diseño del algoritmo

```
algoritmo ejercicio_4_19
var
  real : núm
inicio
  repetir
    leer(archivo,núm)
    si núm <> 0  entonces
      escribir(núm ^ 2)
    fin_si
  hasta_que núm = 0
fin
```

4.20. Algoritmo que reciba una fecha por teclado - dd, mm, aaaa -, así como el día de la semana que fue el primero de enero de dicho año, y muestre por pantalla el día de la semana que corresponde a la fecha que le hemos dado. En su resolución deben considerarse los años bisiestos.

Análisis del problema

Se comenzará realizando una depuración en cuanto a la introducción de datos. Una vez aceptada una fecha como correcta, se inicializará a 0 la variable `dtotal` y se acumulan en ella los días correspondientes a cada uno de los meses transcurridos, desde enero hasta el mes anterior al que indica la fecha, ambos inclusive. Por último le añadiremos los días del mes actual.

Si hacemos grupos de 7 con `dtotal`, se obtendrá un cierto número de semanas y un resto. Para averiguar el día de la semana en que cae la fecha especificada, bastará con avanzar, a partir del día de la semana que fue el primero de enero de dicho año, el número de días que indica el resto.

Se ha de tener en cuenta que, considerando que la semana comienza en lunes, avanzar desde el domingo consiste en volver a situarse en el lunes.

Diseño del algoritmo

```
algoritmo  ejercicio_4_20
var
    entero : dd, mm, aaaa, contm, dtotal, d1
    carácter : día
    lógico   : correcta, bisiesto
inicio
  repetir
    bisiesto ← falso
```

```
   correcta ← verdad
   escribir('Deme fecha (dd mm aaaa) ')
   leer(dd, mm, aaaa)
   si (aaaa mod 4 = 0) y (aaaa mod 100 <> 0) o
         ➥(aaaa mod 400 = 0) entonces
     bisiesto ← verdad
   fin_si
   según_sea mm hacer
     1, 3, 5, 6, 8, 10, 12:
       si dd > 31 entonces
         correcta ← falso
       fin_si
     4, 7, 9, 11:
       si dd > 30 entonces
         correcta ← falso
       fin_si
     2:
       si bisiesto entonces
         si dd > 29 entonces
           correcta ← falso
         fin_si
       si_no
         si dd > 28 entonces
           correcta ← falso
         fin_si
       fin_si
   si_no
     correcta  ← falso
   fin_según
hasta_que correcta
dtotal ← 0
desde contm ← 1 hasta mm - 1 hacer
   según_sea contm hacer
     1, 3, 5, 6, 8, 10, 12:
       dtotal ← dtotal + 31
     4, 7, 9, 11:
       dtotal ← dtotal + 30
     2:
     si bisiesto entonces
       dtotal ← dtotal + 29
     si_no
       dtotal ← dtotal + 28
     fin_si
   fin_según
fin_desde
dtotal ← dtotal + dd
repetir
   escribir('Deme día de la semana que fue el 1
         ➥de ENERO (l/m/x/j/v/s/d) ')
   leer(día)
```

```
      según_sea día hacer
         'l','L':
            d1 ← 0
         'm','M':
            d1 ← 1
         'x','X':
            d1 ← 2
         'j','J':
            d1 ← 3
         'v','V':
            d1 ← 4
         's','S':
            d1 ← 5
         'd','D':
            d1 ← 6
      si_no
         d1 ← -10
      fin_según
   hasta_que (d1 >= 0) y (d1 <= 6)
   dtotal ← dtotal + d1
   según_sea dtotal mod 7 hacer
      1:
         escribir('Lunes')
      2:
         escribir('Martes')
      3:
         escribir('Miércoles')
      4:
         escribir('Jueves')
      5:
         escribir('Viernes')
      6:
         escribir('Sábado')
      0:
         escribir('Domingo')
   fin_según
fin
```

5

Subprogramas (subalgoritmos), procedimientos y funciones

5.1. Programación modular

El diseño descendente resuelve un problema efectuando descomposiciones en otros problemas más sencillos a través de distintos niveles de refinamiento. La programación modular consiste en resolver de forma independiente los subproblemas resultantes de una descomposición.

La programación modular completa y amplía el diseño descendente como método de resolución de problemas y permite proteger la estructura de la información asociada a un subproblema.

Cuando se trabaja de este modo, existirá un algoritmo principal o conductor que transferirá el control a los distintos módulos o subalgoritmos, los cuales, cuando terminen su tarea, devolverán el control al algoritmo que los llamó. Los *módulos* o *subalgoritmos* deberán ser pequeños, seguirán todas las reglas de la programación estructurada y podrán ser representados con las herramientas de programación habituales.

El empleo de esta técnica facilita notoriamente el diseño de los programas; por ejemplo:

- Como los módulos son independientes, varios programadores podrán trabajar simultáneamente en la confección de un algoritmo, repartiéndose las distintas partes del mismo.
- Se podrá modificar un módulo sin afectar a los demás.
- Las tareas, subalgoritmos, sólo se escribirán una vez, aunque se necesiten en distintas ocasiones

a lo largo del algoritmo.

Existen dos tipos de subalgoritmos: *funciones* y *procedimientos.*

5.2. Funciones

Una función toma uno o más valores, denominados argumentos o parámetros actuales y, según el valor de éstos, devuelve un resultado en el nombre de la función. Para invocar a una función se utiliza su nombre seguido por los parámetros actuales o reales entre paréntesis en una expresión. Es decir que se podrá colocar la llamada a una función en cualquier instrucción donde se pueda usar una expresión. Por ejemplo **escribir**(**raíz2**(16)). O, si la función se denomina `f` y sus parámetros son `p1`, `p2` y `p3` **escribir**(f(p1,p2,p3)).

Cada lenguaje de programación tiene sus propias funciones incorporadas, que se denominan internas o intrínsecas. Se considerarán como internas únicamente las más básicas y comunes a casi todos los lenguajes y se irán comentando a lo largo del libro en los capítulos adecuados, es decir cuando para explicar el tema se necesite una referencia a alguna de ellas.

Si las funciones estándar no permiten realizar el tipo de cálculo deseado será necesario recurrir a las funciones externas, que definiremos mediante una declaración de función.

5.2.1. Declaración de funciones

Las funciones, como subalgoritmos que son, tienen una constitución similar a los algoritmos. Por consiguiente, una función constará de:

- cabecera, con la definición de la función
- cuerpo de la función

Dentro del cuerpo de la función estará el bloque de declaraciones y el bloque de instrucciones. Este debe incluir una instrucción mediante la que la función tomará un valor para devolverlo al algoritmo llamador.

Para que las acciones descritas en una función sean ejecutadas se necesita que ésta sea invocada, y se le proporcionen los argumentos necesarios para realizar esas acciones. En la definición de la función deberán figurar una serie de parámetros, denominados parámetros formales, para que, cuando se llame a la función se pueda establecer una correspondencia uno a uno y de izquierda a derecha entre los parámetros actuales y los formales. En el cuerpo de la función se utilizarán los parámetros formales cuando se quiera trabajar con información procedente del programa llamador. El pseudocódigo correspondiente a una función sería

```
<tipo_de_dato> función <nombre_función>(lista_de_parámetros_formales)
   [ declaraciones locales ]
inicio
   ......
   ......
   devolver(<expresión>)
fin_función
```

La *lista_de_parámetros_formales* estará formada por una o más sublistas de parámetros de la siguiente forma:

```
{E|S|E/S}<tipo_de_dato>:<nombre_de_parámetro_formal>...
```

Las llaves quieren decir que se elija sólo una entre las distintas opciones que aparecen separadas por una barra. En las funciones habitualmente será **E**. Todo esto se detallará más adelante. Los corchetes indican no obligatoriedad. El tipo de dato debe ser estándar o haber sido definido de antemano. Podemos separar distintos tipos de parámetros utilizando el punto y coma (;) entre cada declaración.

5.3. Procedimientos

Un procedimiento es un subalgoritmo que realiza una tarea específica y que puede ser definido con 0, 1 o N parámetros. Tanto la entrada de información al procedimiento como la devolución de resultados desde el procedimiento al programa llamador se realizarán a través de los parámetros. El nombre de un procedimiento no está asociado a ninguno de los resultados que obtiene.

La invocación a un procedimiento se realiza con una instrucción llamar_a o bien directamente con el nombre del procedimiento. Es decir:

```
[llamar_a] <nombre_procedimiento> [(lista_de_parámetros_actuales)]
```

Como se puede apreciar con respecto a la lista de parámetros no existe obligatoriedad.

5.3.1. Declaración de procedimientos

La declaración de un procedimiento es similar a la de una función, las pequeñas diferencias son debidas a que el nombre del procedimiento no se encuentra asociado a ningún resultado. La declaración de un procedimiento expresada en pseudocódigo sería:

```
procedimiento <nombre_procedimiento>[(lista_de_parámetros_formales )]
[ declaraciones locales ]
inicio
   ......
   ......
   ......
fin_procedimiento
```

La *lista_de_parámetros_formales* estará formada por una o más sublistas de parámetros de la siguiente forma:

```
{E|S|E/S}<tipo_de_dato>:<nombre_de_parámetro_formal>...
```

y seguiría la mismas reglas que la declaración de parámetros en las funciones.

En el algoritmo principal las declaraciones de procedimientos y funciones, se situarán al final, al objeto de agilizar la escritura de algoritmos.

5.4. Estructura general de un algoritmo

```
algoritmo <nombre_algoritmo>
const
   <nombre_de_constante1>=valor1
   .............................
tipo
   <clase>: <nombre_del_tipo>     // Se verá en capítulos posteriores
var
   <nombre_del_tipo>:<nombre_de_variable1>[,<nombre_de_variable2>,...]
   ...............................................................
   //Los datos han de ser declarados antes de poder ser utilizados
inicio
   <acción1>
   <acción2>
   llamar_a<nombre_de_procedimiento>[(lista_de_parámetros_actuales)]
   // la llamada a la función ha de realizarse en una expresión
   escribir(<nombre_de_función> (lista_de_parámetros_actuales))
   //Se utilizará siempre la sangría en las estructuras
   //selectivas y repetitivas.
   .........
   <acciónN>
fin

procedimiento <nombre_procedimiento>[(lista_de_parámetros_formales)]
[ declaraciones locales ]
inicio
   ......
   ......
   ......
fin_procedimiento

<tipo_de_dato> función <nombre_función>[(lista_de_parámetros_formales)]
// el <tipo_de_dato> devuelto por la función es un tipo estándar
[ declaraciones locales ]
inicio
   ......
   ......
   devolver(<expresión>)
fin_función
```

5.5. Paso de parámetros

Cuando un programa llama a un procedimiento o función se establece una correspondencia entre los parámetros actuales y los formales. Existen dos formas para establecer la correspondencia de parámetros:

- **Posicional**. Emparejando los parámetros reales y formales según su posición en las listas. Esto

requiere que ambas listas tengan el mismo número de parámetros y que los que se van a emparejar coincidan en el tipo. En la definición del subprograma deberá reflejarse siempre de qué tipo es cada uno de los parámetros formales.

```
(E <tipo_de_dato>:<nombre1_de_parámetro_formal>...)
```

El tipo de dato debe ser estándar o haber sido definido de antemano. Si los parámetros formales se separan por comas es necesario, aunque no suficiente, que tengan el mismo tipo. Si su tipo fuera distinto habría que poner:

```
(E <tipo_de_dato1>: <nombre1_de_parámetro_formal> ;
 E  <tipo_de_dato2>: <nombre2_de_parámetro_formal>)
```

- **Correspondencia por el nombre explícito.** En las llamadas se indica explícitamente la correspondencia entre los parámetros reales y formales.

Dado que la mayor parte de los lenguajes usan exclusivamente la correspondencia posicional, éste será el método que se seguirá en la mayoría de los algoritmos.

Al hablar de los procedimientos se decía que devuelven resultados al programa principal a través de los parámetros, pero que también pueden recibir información, desde el programa principal, a través de ellos. Esto nos lleva a una clasificación de los parámetros en:

Parámetros de entrada Permiten únicamente la transmisión de información desde el programa llamador al subprograma.

Parámetros de salida Sólo devuelven resultados.

Parámetros de entrada/salida Actúan en los dos sentidos, tanto mandando valores al subprograma, devolviendo resultados desde el subprograma al programa llamador.

En los algoritmos, se debe especificar en la definición del subprograma cómo se desea que se comporte cada uno de los parámetros; para ello se empleará la siguiente terminología:

- **E** equivaldría a parámetro de entrada
- **S** querrá decir parámetro de salida
- **E/S** parámetro de entrada/salida

En la lista de parámetros siguiente

```
(E  <tipo_de_dato1>: <nombre1_de_parámetro_formal> ;
 S <tipo_de_dato1>: <nombre2_de_parámetro_formal>)
```

<nombre1_de_parámetro_formal> es parámetro de entrada y va a proporcionar datos al subprograma. *<nombre2_de_parámetro_formal>* es parámetro de salida y devolverá resultados al programa llamador. Aunque ambos son del mismo tipo, *<tipo_de_dato1>*, habrá que repetir el tipo para cada uno de ellos y no se escriben separados por coma.

Todo esto afectará tanto a procedimientos como a funciones. De lo que se deduce que una función va a tener la posibilidad de devolver valores al programa principal de dos formas:

- Como valor de la función
- A través de los parámetros

Un procedimiento sólo podrá devolver resultados a través de los parámetros, de modo que al codificar el algoritmo se ha de tener mucho cuidado con el paso de parámetros, siendo preciso conocer los métodos de transmisión que posee el lenguaje, para poder conseguir el funcionamiento deseado. Los

lenguajes suelen disponer de:

Paso por valor Los parámetros formales correspondientes reciben una copia de los valores de los parámetros actuales; por tanto los cambios que se produzcan en ellos por efecto del subprograma no podrán afectar a los parámetros actuales y no se devolverá información al programa llamador. Los parámetros resultarían de Entrada, **E**.

Paso por valor resultado Al finalizar la ejecución del subprograma los valores de los parámetros formales se transfieren o copian a los parámetros actuales.

Paso por referencia Lo que se pasa al procedimiento es la dirección de memoria del parámetro actual. De esta forma, una variable pasada como parámetro actual es compartida; es decir, se puede modificar directamente por el subprograma. Los parámetros serían de Entrada/Salida, **E/S**.

Es posible pasar como parámetros datos y subprogramas.

5.6. Variables globales y locales

Una variable es global cuando el ámbito en el que dicha variable se conoce es el programa completo. Consideraremos como variables globales las que hayan sido declaradas en el programa principal y como locales las declaradas en el propio subprograma.

Toda variable que se utilice en un procedimiento debe haber sido declarada en él. De esta forma todas las variables del procedimiento serán locales y la comunicación con el programa principal se realizará exclusivamente a través de los parámetros. Al declarar una variable en un procedimiento no importa que ya existiera otra con el mismo nombre en el programa principal; ambas serán distintas y, cuando nos encontremos en el procedimiento, sólo tendrá vigencia la declaración que hayamos efectuado en él. Trabajando de esta forma obtendremos la independencia de los módulos.

5.7. Recursividad

Un objeto es recursivo si forma parte de sí mismo o interviene en su propia definición. El instrumento necesario para expresar los programas recursivamente es el subprograma. La mayoría de los lenguajes de programación admiten que un procedimiento o función haga referencia a sí mismo dentro de su definición, recursividad directa. También es posible que un procedimiento o función haga referencia a otro el cual contenga, a su vez, una referencia directa o indirecta al primero, recursividad indirecta.

La recursión se puede considerar como una alternativa a la iteración y, aunque las soluciones iterativas están más cercanas a la estructura de la computadora, resulta muy útil cuando se trabaja con problemas o estructuras, como los árboles, definidos en modo recursivo. Por ejemplo el factorial de un número n, que se define matemáticamente como:

```
n! = n * (n-1)!
```

El problema general se resuelve en términos de otro caso del mismo, hasta llegar a uno que se resuelve de forma no recursiva. Existe un acercamiento paulatino al caso no recursivo. Para comprender la recursividad hay que tener en cuenta que:

- Cuando se llama a un procedimiento o función los parámetros y las variables locales toman nuevos valores y el procedimiento o función trabaja con estos nuevos valores y no con los de las anteriores llamadas.
- Cada vez que se llama a un procedimiento o función los parámetros de entrada y variables locales son almacenados en las siguientes posiciones libres de memoria y cuando termina la ejecución del procedimiento o función son accedidos en orden inverso a como se introdujeron.
- El espacio necesitado para almacenar los valores crece pues conforme a los niveles de anidamiento de las llamadas.
- El cuerpo del procedimiento o función debe disponer de una o varias instrucciones selectivas donde establecer la condición o condiciones de salida.

Todo algoritmo recursivo puede ser implementado en forma iterativa utilizando una pila.

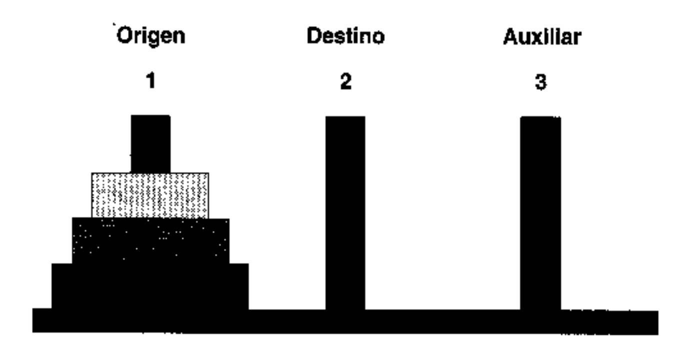
5.7.1. Algoritmos recursivos

El algoritmo de Ordenación Rápida (Quick Sort) tiene una definición recursiva. Ordena un array dividiéndolo en dos particiones más pequeñas del mismo, de forma que todos los elementos de una de ellas son menores que cada uno de los de la segunda y todos los de la otra mayores que cada uno de los de la primera. Estas particiones se subdividirán por separado hasta obtener particiones de un sólo elemento, terminación del proceso recursivo. Este algoritmo se desarrollará más adelante, cuando se trate el apartado de ordenación interna.

La forma iterativa de este algoritmo se verá en el capítulo que estudia los métodos de ordenación interna. Dejamos al lector la comparación entre dicha forma iterativa y la recursiva que aparece desarrollada en los ejercicios del presente capítulo.

El problema de las Torres de Hanoi es un problema clásico de recursión. Se tienen 3 torres y un conjunto de discos de diferentes tamaños. Cada disco tiene una perforación en el centro que le permite ensartarse en cualquiera de las torres. Los discos han de encontrarse siempre situados en alguna de las torres. Inicialmente todos están en la misma torre, ordenados de mayor a menor, como se muestra en el dibujo. Se deben averiguar los movimientos necesarios para pasar todos los discos a otra torre, utilizando la tercera como auxiliar y cumpliendo las siguientes reglas:

- En cada movimiento sólo puede intervenir un disco.
- No puede quedar un disco sobre otro de menor tamaño.



Para N = 3 la solución del problema implicaría los siguientes movimientos

1. De 1 a 2
2. De 1 a 3
3. De 2 a 3
4. De 1 a 2
5. De 3 a 1
6. De 3 a 2
7. De 1 a 2

```
algoritmo Hanoi
var
  entero: n
inicio
  escribir('Nº de discos:')
  leer(n)
  llamar_a Mover_torre(n,1,2,3)
fin

procedimiento Mover_torre( E entero: N ;  E entero: orig,dest,aux)
inicio
  si N = 1 entonces
    escribir('Paso de ', orig,' a ', dest )
  si_no
    llamar_a Mover_torre ( N-1, orig, aux, dest )
    escribir('Paso de ', A,' a ', B )
    llamar_a Mover_torre ( N-1, aux, dest, orig )
  fin_si
fin_procedimiento
```

5.7.2. Algoritmos con retroceso

Hay un gran número de problemas cuya solución no puede obtenerse mediante una serie de cálculos más o menos complejos y su resolución requiere la aplicación de métodos de tanteo de forma sistemática, es decir ir probando todas las opciones posibles. Existen dos posibilidades: que la opción escogida forme parte de la solución buscada o que no, en cuyo caso se retrocederá para probar con otra posible opción. En estos problemas resulta muy útil la recursividad.

Un ejemplo de este tipo es el problema de las ocho reinas, que consiste en disponer ocho reinas en un tablero de ajedrez de tal forma que no se amenacen entre sí (recuerde que una reina amenaza a las piezas situadas en su misma fila, columna o diagonal). La tarea más importante a realizar en este algoritmo, cuyo pseudocódigo se incluye en un ejercicio al final de este capítulo, es seleccionar adecuadamente la estructura de datos para representar el problema y los parámetros de los procedimientos recursivos.

5.8. Ejercicios resueltos

5.1. Realizar un procedimiento que permita intercambiar el valor de dos variables.

Análisis del Problema

Para intercambiar el contenido de dos variables, es necesaria una variable auxiliar, del mismo tipo de datos que las otras variables. Se pasan como parámetros de entrada las dos variables cuyo valor se desea intercambiar. Ya que su valor será modificado durante la ejecución del subalgoritmo, serán parámetros de entrada/salida y el subalgoritmo será un procedimiento.

Diseño del algoritmo

```
procedimiento Intercambio(E/S entero :  a,b)
var
  entero : aux
inicio
  aux ← a
  a ← b
  b  ← aux
fin_procedimiento
```

Este procedimiento sólo serviría para intercambiar variables de tipo entero. Se podría hacer un procedimiento más genérico utilizando un tipo de datos abstracto. En la cabecera del programa principal podemos definir un tipo de datos `TipoDatos` de la forma

```
tipo
  TipoDatos = .... // tipo de datos de las variables
                   // a intercambiar
```

y modificar la cabecera del procedimiento,

```
procedimiento Intercambio(E/S TipoDatos :  a,b)
var
  TipoDatos: aux
```

5.2. Realizar una función no recursiva que permita obtener el término n de la serie de Fibonacci.

Análisis del problema

La serie de Fibonacci se define como:

$\text{Fibonacci}_n = \text{Fibonacci}_{n-1} + \text{Fibonacci}_{n-2}$ para todo $n > 1$

para $n <= 1$, $\text{Fibonacci}_n = 1$.

Por lo tanto se deben sumar los elementos mediante un bucle que debe ejecutarse desde 2 hasta `n`. Cada iteración debe guardar el último y el penúltimo término, para lo que se utilizan dos variables `último` y `penúltimo`, a las que irán cambiando sus valores. El subalgoritmo aceptará una como entrada una variable entera y devolverá un valor también entero, por lo que deberemos utilizar una función.

Diseño del algoritmo

```
entero función Fibonacci(E entero : n)
var
   entero : i, último, penúltimo, suma
inicio
   suma ← 1
   último ← 1
   penúltimo ← 1
   desde i ← 2 hasta n hacer
      penúltimo ← último
      último ← suma
      suma  ← último + penúltimo
   fin_desde
   devolver(suma)
fin_función
```

5.3. *Implementar una función que permita devolver un valor entero, leído desde teclado, comprendido entre dos límites que introduciremos como parámetro.*

Análisis de problema

Esta función puede ser útil para validar una entrada de datos de tipo entero y se podrá incluir en otros algoritmos. Consistirá simplemente en un bucle **`repetir`** que ejecutará la lectura de un dato hasta que esté entre los valores que se han pasado como parámetros de entrada.

Diseño del algoritmo

```
entero función ValidarEntero(E entero : inferior, superior)
var
   entero : n
inicio
   repetir
      leer(n)
   hasta_que n >= inferior y n <= superior
   devolver(n)
fin_función
```

Los parámetros `inferior` y `superior` deberán pasarse en dicho orden; es decir, primero el menor y luego el mayor, aunque con una pequeña modificación podrían pasarse en cualquier orden. Para ello debería incluirse una condición antes de comenzar el bucle que, de ser necesario, haría una llamada al procedimiento `Intercambio`, ya desarrollado:

```
.
.
si inferior > superior entonces
   Intercambio(inferior, superior)
fin_si
.
.
```

5.4. Diseñar una función que permita obtener el valor absoluto de un número.

Análisis del problema

El valor absoluto de un número positivo, sería el mismo número; de un número negativo sería el mismo número sin el signo y de 0 es 0. Por lo tanto, esta función, únicamente debería multiplicar por -1 el número pasado como parámetro de entrada en el caso que éste fuera menor que 0.

Diseño del algoritmo

```
entero función Abs(E entero : n)
inicio
   si n < 0 entonces
      devolver(n * -1)
   si_no
      devolver(n)
   fin_si
fin_función
```

Esta función sólo es válida para números enteros. Para que valiera con algún otro tipo de dato numérico, deberíamos utilizar un tipo de dato abstracto.

5.5. Realizar un procedimiento que obtenga la división entera y el resto de la misma utilizando únicamente los operadores suma y resta.

Análisis del problema

La división se puede considerar como una sucesión de restas. El algoritmo trata de contar cuántas veces se puede restar el divisor al dividendo y dicho contador sería el cociente. Cuando ya no se pueda restar más sin que salga un número positivo, se tendrá el resto.

Por lo tanto se tienen dos parámetros de entrada, `dividendo` y `divisor`, y dos de salida, `cociente` y `resto`. `cociente` será un contador que se incrementará en 1 cada vez que se pueda restar el `divisor` al `dividendo`. El bucle a utilizar será de tipo **mientras**, ya que puede darse el caso que no se ejecute ninguna vez, en cuyo caso el cociente será 0 y el resto será el dividendo.

Diseño del algoritmo

```
procedimiento DivisiónEntera( E entero : dividendo, divisor ;
                              S entero : cociente, resto)
inicio
   cociente ← 0
   mientras dividendo => divisor hacer
      dividendo ← dividendo - divisor
      cociente ← cociente + 1
   fin_mientras
   resto ← dividendo
fin_procedimiento
```

5.6. Diseñar un procedimiento que permita convertir coordenadas polares (radio, ángulo) en cartesianas (x,y)

*x = radio * cos(ángulo)*
y = radio sen(ángulo)*

Análisis del problema

La resolución requiere aplicar la fórmula indicada más arriba. Habrá que tener en cuenta el tipo de parámetros. `radio` y `ángulo` serán de entrada y `x` e `y` de salida.

Diseño del algoritmo

```
procedimiento Polares( E real : ángulo, radio ; S real : x, y)
inicio
   x ← radio * cos(ángulo)
   y ← radio * sen(ángulo)
fin_procedimiento
```

5.7. Diseñe una función que permita obtener el factorial de un número entero positivo.

Análisis del problema

El factorial de *n* se puede definir para cualquier entero positivo como

$$Factorial_n = n * n-1 * n-2 * .....* 1$$

por definición, $Factorial_0 = 1$

Por lo tanto para implementar un función `Factorial`, se deberá realizar un bucle que se ejecute entre 2 y *n*, acumulando en su cuerpo las sucesivas multiplicaciones.

Diseño del algoritmo

```
entero función Factorial( E entero : n)
var
  entero : i, f
inicio
  f ← 1
  desde i ← 2 hasta n hacer
    f ← f * i
  fin_desde
  devolver(f)
fin_función
```

5.8. *Diseñar una función que permita obtener el máximo común divisor de dos números mediante el algoritmo de Euclides.*

Análisis del problema

Para obtener el máximo común divisor de dos números enteros positivos *a* y *b* según el algoritmo de Euclides, se debe ejecutar un bucle que divida *a* entre *b*. Si el resto es 0, *b* será el divisor; en caso contrario, *a* tomará el valor de *b* y *b* el del resto de la división anterior. El bucle finalizará cuando el resto sea 0, es decir, cuando *a* sea divisible entre *b*.

Diseño del algoritmo

```
entero función Mcd (E entero :a,b)
var
  entero : resto
inicio
  mientras a mod b <> 0 hacer
    resto ← a mod b
    a  ← b
    b  ← resto
  fin_mientras
```

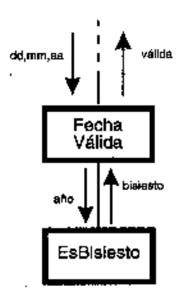
```
   devolver(b)
fin_función
```

5.9. Realizar una función que permita saber si una fecha es válida.

Análisis del problema

Esta función es muy normal en cualquier aplicación informática. Se trata de ver si una fecha, introducida como *mes*, *día* y *año* es una fecha correcta; es decir, el día ha de estar comprendido entre 1 y 31, el mes entre 1 y 12, y, si esto es correcto, ver si el número de días para un mes concreto es válido.

El tipo de la función será un valor lógico, verdadero si la fecha es correcta o falso si es errónea. Como ayuda, se utilizará una función `EsBisiesto`, a la que se pasará el valor entero correspondiente a un año determinado y devolverá un valor lógico verdadero si el año el bisiesto y falso en caso contrario. Un año será bisiesto si es divisible por 4, excepto los que son divisibles por 100 pero no por 400, es decir, menos aquellos con los que comienza el siglo. Un diseño modular de la función podría ser por tanto:



Diseño del algoritmo

```
lógico función FechaVálida(E entero : dd,mm,aa)
inicio
   FechaVálida ← verdad
   si (mm < 1) o (mm > 12) entonces
      devolver(falso)
   si_no
      si dd < 1 entonces
         devolver(falso)
      si_no
         según_sea mm hacer
            4,6,9,11 : si dd > 30 entonces
                          devolver(falso)
                       fin_si
            2 : si Esbisiesto(aa) y (dd > 29) entonces
                   devolver(falso)
```

```
              si_no
                    si no Esbisiesto(aa) y (dd > 28) entonces
                         devolver(falso)
                   fin_si
             fin_si
        si_no
          si dd > 31 entonces
            devolver(falso)
          fin_si
        fin_según
      fin_si
   fin_si
fin_función

lógico función EsBisiesto(E entero :  aa)
inicio
   devolver((aa mod 4 = 0) y (aa mod 100 <> 0) o (aa mod 400 = 0))
fin_función
```

5.10. Implementar una función que permita hallar el valor de X^y*, siendo* X *un número real e* y *un entero.*

Análisis del problema

Se trata de una función similar a la del factorial, pues se trata de acumular multiplicaciones, pues debemos multiplicar X por sí mismo y veces. Si y es negativo, X^y es igual a su inversa.

Diseño del algoritmo

```
real función Potencia( E entero :  x,y)
var
   entero : i, p
inicio
   p ← 1
   desde i ← 1 hasta abs(y) hacer
      p ← p * x
   fin_desde
   si y < 0 entonces
      devolver(p)
   si_no
      devolver(1/p)
   fin_si
fin_función
```

5.11. Realizar tres funciones que permitan hallar el valor de π *mediante las series matemáticas siguientes:*

a) $\pi = 4\sum_{i=1}^{\infty}\frac{(-1)^i}{2i+1} = 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots\right)$

b) $\pi = \frac{1}{2}\sqrt{\sum_{i=1}^{\infty}\frac{24}{(i)^2}} = \frac{1}{2}\sqrt{24+\frac{24}{2^2}+\frac{24}{3^2}+\frac{24}{4^2}+\frac{24}{5^2}+\ldots}$

c) $\pi = 4\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\ldots$

La precisión del cálculo dependerá del número de elementos de la serie, n, *que será un parámetro que se pase a la función.*

Análisis del problema

En cualquiera de los tres casos se debe implementar un bucle que se ejecute n veces y en el que se irá acumulando la suma de los términos de la serie.

En el caso (*a*), los términos pares suman y los impares restan, por lo que será preciso hacer una distinción. El numerador siempre es 1, y el denominador son los *n* primeros números impares. En el caso (*b*), el numerador es 24 y el denominador será el cuadrado de la variable del bucle. El resultado final será la raíz cuadrada de la serie. En el caso (*c*) lo que se debe acumular son los productos; el numerador serán los números pares y el denominador los impares.

Diseño del algoritmo

(a)

```
real función Pi( E entero : n)
var
   real : serie
   entero : i, denom
   lógico : par
inicio
   denom ← 1
   serie ← 0
   par ← falso
   desde i ← 1 hasta n hacer
      si par entonces
         serie ← serie - 1 / denom
      si_no
         serie ← serie + 1 / denom
      fin_si
      par ← no par
      denom ← denom + 2
   fin_desde
   devolver(serie * 4)
fin_función
```

(b)

```
real función Pi(E entero : n)
var
  real : serie
  entero : i
inicio
  serie ← 0
  desde i ← 1 hasta n hacer
    serie ← serie + 24 / i^2
  fin_desde
  devolver(raíz2(serie) / 2)
fin_función
```

(b)

```
real función Pi(E entero : n)
var
  real : serie
  entero : i
inicio
  serie ← 2
  desde i ← 1 hasta n hacer
    si i mod 2 = 0 entonces
      serie ← serie * i / (i + 1)
    si_no
      serie ← serie * (i + 1) / i
    fin_si
  fin_desde
  devolver(serie)
fin_función
```

5.12. Realizar un subprograma que calcule la suma de los divisores de n distintos de n.

Análisis del problema

Para obtener los divisores del número entero n, será preciso hacer un bucle en el que un contador se irá decrementando desde n-1 hasta 1. Por cada iteración, se comprobará si el contador es divisor de n. Como el primer divisor que podemos encontrar será n/2, el valor inicial del contador podrá ser n **div** 2.

El tipo de subprograma deberá ser una función, a la que pasaríamos como parámetro de entrada el número n.

Diseño del algoritmo

```
entero función SumaDivisor(E entero :  n)
```

```
var
   entero : suma,i
inicio
   suma ← 0
   desde i ← n div 2 hasta 1 incremento -1 hacer
      si n mod i = 0 entonces
         suma ← suma + i
      fin_si
   fin_desde
   devolver(suma)
fin_función
```

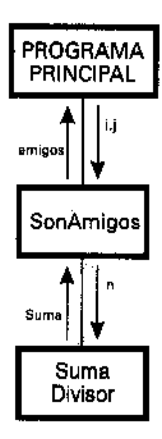
5.13. *Dos números son* amigos, *si cada uno de ellos es igual a la suma de los divisores del otro. Por ejemplo, 220 y 284 son amigos, ya que:*

Suma de divisores de 284 : 1+2+4+71+142 = 220
Suma de divisores de 220: 1+2+4+5+10+11+20+22+44+55+110 = 284

Diseñe un algoritmo que muestre todas las parejas de números amigos menores o iguales que m, siendo m un número introducido por teclado.

Análisis del problema

Este algoritmo se puede descomponer en tres partes diferenciadas. Por una parte un programa principal que debe ir sacando todas las parejas de números mayores o iguales a un número m que previamente habremos introducido por teclado. Desde ahí se llamará a una función lógica `SonAmigos` que dirá si la pareja de números son amigos. Para hacer esto, se calcula la suma de los divisores, para lo que llamaremos a la función `SumaDivisor`, desarrollada más arriba. Por lo tanto un diseño modular del programa podría ser como sigue:



El programa principal leerá el número m, e implementará dos bucles anidados para sacar las parejas menores o iguales que m. Para ello, un primer bucle realizará la iteraciones de i entre 1 y m, mientras que el bucle interno irá tomando valores entre i+1 y m. Dentro del bucle habrá una llamada

a la función `SonAmigos`, y si ésta devuelve un valor verdadero escribiremos la pareja. La función `SonAmigos` se encarga de comprobar si una pareja de números son amigos. Recibe dos parámetros de entrada, y guarda en dos variables la suma de sus divisores mediante la función `SumaDivisor`. Si ambas sumas son iguales, devolverá un valor verdadero; en caso contrario, falso.

Diseño del algoritmo

```
algoritmo ejercicio_5_13
var
   entero : i,j,m
inicio
   leer(m)
   desde i ← 1  hasta  m-1 hacer
      desde j ← i+1 hasta m hacer
         si SonAmigos(i,j) entonces
            escribir(i,j)
         fin_si
      fin_desde
   fin_desde
fin
lógico función SonAmigos( E entero : n,m)
inicio
   devolver((SumaDivisor(n) = m) y (SumaDivisor(m) = n)
fin_función
```

5.14. *El número de combinaciones de* m *elementos tomados de* n *en* n *es:*

$$\binom{m}{n} = \frac{m!}{n!(m-n)}$$

Diseñar una función que permita calcular el número combinatorio $\binom{m}{n}$

Análisis del problema

La función se reduce a una simple asignación, siempre que se tenga resuelto el problema de hallar el factorial, cuya función ya se ha diseñado más arriba. Por lo tanto se limitará a codificar la expresión del número combinatorio.

Diseño del algoritmo

```
entero función Combinatorio(E entero : m,n)
inicio
   devolver(Factorial(m) div Factorial(n) * Factorial(m - n))
fin_función
```

5.15. Implemente tres funciones que permitan averiguar los valores de e^x, cos(x) *y* sen(x) *a partir de las series siguientes:*

$$e^x = \sum_{i=1}^{n} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

$$\cos(x) = 1 + \sum_{i=1}^{n} (-1) \frac{x^{2i}}{(2i)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$

$$\text{sen}(x) = \sum_{i=1}^{n} (-1) \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

El número de términos de la serie será el suficiente para que la diferencia absoluta entre dos valores sucesivos sea menor 10^{-3}.

Análisis del problema

En ambos casos se trata de hacer un bucle que se ejecute hasta que el valor absoluto de la diferencia entre el último y el penúltimo término sea menor que 0.001. Dentro del bucle habrá un acumulador que sume cada uno de los términos.

Para el caso de e^x, en cada término el numerador será x elevado al número de orden del término, mientras que el denominador será el factorial del número de orden del término. El contador del bucle por lo tanto irá tomando valores entre 1 y n.

Para *cos(x)*, el numerador irá elevando x a las potencias pares, mientras que el denominador será el factorial de los números pares. Por lo tanto el contador del bucle irá sacando los números pares. Además, en él los términos de orden par restan, mientras que los de orden impar suman.

Por último, para *sen(x)*, la diferencia estriba en que la potencia y el factorial serían de los números impares, por lo que el contador del bucle sacará los número impares.

En los tres casos, como se ha dicho más arriba, tendremos que utilizar una variable suma que acumulará el valor de la serie, un contador, con las características que se han indicado y dos variables reales para guardar el último y el penúltimo término, para ver si la diferencia entre ambos es menor que 0.001.

Diseño del algoritmo

e^x

```
real función exponente( E entero : x)
var
   entero : i
   real : suma, último, término
inicio
```

```
   suma ← 1 + x
   i ← 1
   término ← x
   repetir
      i ← i + 1
      último ← término
      término ← x^i / Factorial(i)
      suma ← suma + término
   hasta_que abs(término - último) < 0.001
   devolver(suma)
fin_función
```

cos(x)

```
real función cos( E entero : x )
var
   entero : i
   real : suma, último, término
   lógico :par
inicio
   suma ← 1
   término ← 1
   i ← 0
   par ← verdad
   repetir
      i ← i + 2
      último ← término
      término ← x^i / Factorial(i)
      si par entonces
         suma ← suma - término
      si_no
         suma ← suma + término
      fin_si
      par ← no par
   hasta_que abs(término - último) < 0.001
   devolver(suma)
fin_función
```

sen(x)

```
real función sen( E entero : x)
var
   entero : i
   real  : suma, último, término
   lógico : par
inicio
   suma ← x
```

```
   término ← x
   i ← 1
   par ← verdad
   repetir
      i ← i + 2
      último ← término
      término ← x^i / Factorial(i)
      si par entonces
         suma ← suma - término
      si_no
         suma ← suma + término
      fin_si
      par ← no par
   hasta_que abs(término - último) < 0.001
   devolver(suma)
fin_función
```

5.16. *Implementar una función* `Redondeo(a,b)`, *que devuelva el número real* a *redondeado a* b *decimales.*

Análisis del problema

Para redondear un número real *a* a *b* decimales –siendo *b* un número entero positivo–, hay que recurrir a la función estándar **ent** que elimina los decimales de un número real. Si se quiere dejar una serie de decimales, hay que obtener el entero de la multiplicación del número *a* por 10^b y dividir el resultado por 10^b. De esta forma, el resultado, siendo *a*=3.546 y *b*=2, sería:

```
ent(3.546*100)/100 =  ent(354.6)/100 = 354/100 = 3.54
```

Con esto se consigue truncar, no redondear. Para redondear, de forma que hasta 0.5 redondee por defecto y a partir de ahí por exceso, se debe sumar a la multiplicación 0.5:

```
ent(3.546*100+0.5)/100 =  ent(354.6+0.5)/100 =
        ent(355.1)/100 = 355/100 = 3.55
```

Para realizar esta función, únicamente se ha de aplicar la expresión anterior.

Diseño del algoritmo

```
entero función Redondeo( E real : a  ; E entero : b)
inicio
   devolver(ent(a * 10^b + 0.5) / 10^b)
fin_función
```

5.17. Algoritmo que transforma un número introducido por teclado en notación decimal a romana. El número será entero y positivo y no excederá de 3000.

Análisis del problema

Para pasar un número desde notación decimal a romana se transformará individualmente cada uno de sus dígitos. El procedimiento para transformar un dígito es siempre el mismo

Dígito	*Escribir*
de 1 a 3	de 1 a 3 veces p1
4	p1 seguido de p2
de 5 a 8	p2 seguido por, de 0 a 3 veces, p1
9	p1 y a continuación p3

Tenga en cuenta que, en las distintas llamadas que se realizan al procedimiento, se pasarán diferentes parámetros:

	Parámetro 1 (p1)	*Parámetro 2 (p2)*	*Parámetro 3 (p3)*
Unidades	'I'	'V'	'X'
Decenas	'X'	'L'	'C'
Centenas	'C'	'D'	'M'
Miles	'M'	' '	' '

Diseño del algoritmo

```
algoritmo Ejercicio_5_17
var
  entero : n, r, dígito
inicio
  repetir
    escribir('Deme número ')
    leer(n)
  hasta_que (n >= 0) y (n <= 3000)
  r ← n
  dígito ← r div 1000
  r ← r mod 1000
  calccifrarom(dígito, 'M', ' ',' ')
  dígito ← r div 100
  r ← r mod 100
  calccifrarom(dígito, 'C', 'D', 'M')
  dígito ← r div 10
```

```
   r ← r mod 10
   calccifrarom(dígito, 'X', 'L', 'C')
   dígito ← r
   calccifrarom(dígito, 'I', 'V', 'X')
fin

procedimiento calccifrarom (E entero: dígito ; E caracter: p1, p2, p3)
var
   entero: j
inicio
   si dígito = 9 entonces
      escribir( p1, p3)
   si_no
      si dígito > 4 entonces
         escribir(p2)
         desde j ← 1 hasta dígito - 5 hacer
            escribir( p1)
         fin_desde
      si_no
         si dígito = 4 entonces
            escribir( p1, p2)
         si_no
            desde j ← 1 hasta dígito hacer
               escribir(p1)
            fin_desde
         fin_si
      fin_si
   fin_si
fin_procedimiento
```

5.18. Escribir una función, INTEGRAL, que devuelva el área del recinto formado por el eje de las X, las rectas x=a y x=b y el arco de curva correspondiente a una función continua, recibida como parámetro, con valores positivos en el intervalo considerado.

Análisis de problema

La función `integral` dividirá el intervalo [a,b] en n subintervalos y considerará n rectángulos con esas bases y cuyas alturas sean el valor de la función recibida como parámetro, f(x), en el punto medio de los subintervalos. La suma de las áreas de todos estos rectángulos constituirá el valor devuelto por `integral`.

```
// tipo real función(E real : x) : func
real función integral (E func : f    ; E real : a, b ; E entero : n)
var
   real   : baserectángulo,altura,x,s
   entero : i
```

```
inicio
  baserectángulo ← (b   a) / n
  x ← a + baserectángulo/2
  s ← 0
  desde i ← 1 hasta n hacer
    altura ← f(x)
    s ← s + baserectángulo * altura
    x ← x + baserectángulo
  fin_desde
  devolver(s)
fin_función
```

5.19. Escribir una función recursiva que calcule el factorial de un número entero positivo.

Análisis del problema

El factorial de un número entero positivo n se puede definir como:

$$n! = n * n-1!$$

para cualquier número mayor que uno, ya que 1! = 1 y 0! = 0. Por lo tanto la condición de salida de la llamada recursiva será cuando n sea menor o igual que 1.

Diseño del algoritmo

```
entero función fact(E entero : n)
inicio
  si n = 0 entonces
    devolver(1)
  si_no
    devolver(n * fact(n-1))
  fin_si
fin_función
```

5.20. Escriba una función recursiva que calcule la potencia de un número entero positivo.

Análisis del problema

Una definición recursiva de x^y para dos números enteros positivos es la siguiente:

$$x^y = x * x^{y-1}$$

(x^0 es 1. Esa será la condición de salida de las llamadas recursivas).

Diseño del algoritmo

```
entero función pot(E entero : x,y)
inicio
   si y = 0 entonces
      devolver(1)
   si_no
      devolver(x * pot(x,y-1)
   fin_si
fin_función
```

5.21. *Escribir una función recursiva que calcule el término n de la serie de Fibonacci.*

Análisis del problema

La serie de Fibonacci es la siguiente,

1 1 2 3 5 8 13 21 34...

es decir, cada número es la suma de los dos anteriores, con excepción de los dos primeros, que siempre son 1. Por lo tanto también podemos hacer una definición recursiva para averiguar el término n de la serie, puesto que

$$Fib_n = Fib_{n-1} + Fib_{n-2}$$

para cualquier n mayor que 2.

Diseño del algoritmo

```
entero función Fib(E entero : n)
inicio
   si n = 0 o n = 1 entonces
      devolver(1)
   si_no
      devolver(Fib(n-1) + Fib(n-2)
   fin_si
fin_función
```

5.22. *Escribir un procedimiento recursivo que escriba un número en base 10 convertido a otra base entre 2 y 9.*

Análisis del problema

Para convertir un número decimal a otra base se debe dividir el número entre la base, y repetir el

proceso, haciendo que el dividendo sea el cociente de la siguiente división hasta que éste sea menor que la base. En ese momento, se recogen todos los restos y el último cociente en orden inverso a como han salido.

Este ejemplo es una muestra de cómo puede ser útil la recursividad cuando se trata de un proceso que debe recoger los resultados en orden inverso a como han salido. Para simplificar el ejercicio se ha limitado la base a 9, de forma que se evite tener que convertir los restos en letras.

Diseño del algoritmo

```
procedimiento convierte(E entero : n,b)
inicio
   si n >= b entonces
         convierte(n div b, b)
   fin_si
   escribir(n mod b)
fin_función
```

5.23. Diseñar un algoritmo que resuelva el problema de las ocho reinas.

Análisis del problema

El problema consiste en colocar las 8 damas dentro del tablero de ajedrez sin que se coman unas a otras, en general, se tratará de poner n damas en un tablero de *n x n*.

En vez de colocar las damas en un *array** de *n x n* elementos, se utiliza un vector de *n* elementos. Cada elemento del vector representa la fila donde está la reina. El contenido representa la columna.

El vector `Damas` se inicializa a 0. Para buscar la siguiente solución se comienza situando en `Damas[1]` un 1 (la primera dama está en la primera fila, columna 1). La siguiente dama estará en la fila 2 y empezamos a buscar en la posición 1. No será una posición válida si `Damas[1] = Damas[2]` (están en la misma columna); `Damas[1]+1 = Damas[2]+2` (están en una diagonal) y `Damas[1]-1 = Damas[2]-2` (está en la otra diagonal). Si esto ocurre, incrementamos la columna de la nueva dama y se volverá a comprobar.

Diseño del algoritmo

```
algoritmo Ejericio_5_24
const n = 8
tipos array de entero[1..n] : ListaDamas
```

* *El concepto de array se verá en el capítulo siguiente.*

```
var
   ListaDamas : D
   entero : i
   lógico : solución
inicio
   // Inicializar el array
   desde i ← 1 hasta n hacer
      d[i] ← 0
   fin_desde
   Ensayar(d,1,solución)
   si no solución entonces
      escribir('No hay solución')
   si_no
      // se deja al lector hacer la presentación del resultado
   fin_si
fin
lógico función PosiciónVálida(E ListaDamas d ;  E entero : i)
var
   entero : j
   lógico : válida
inicio
   válida ← verdad
   desde j ← 1 hasta i - 1 hacer
      // No se ataca en la columna
      válida ← válida y (d[i] <> d[j])
      // no se ataca en una diagonal
      válida ← válida y (d[i] + i <> d[j] + j )
      // no se ataca en la otra diagonal
      válida ← válida y (d[i] - i <> d[j] - j )
   fin_desde
   devolver(Válida)
fin_función

procedimiento Ensayar( E/S ListaDamas : d ; E entero : i ;
                       S lógico   : Solución)
inicio
   si i = n + 1 entonces
      solución ← verdad
   si_no
      solución ← falso
      repetir
         d[i] ← d[i] + 1
         si PosiciónVálida(d,i) entonces
            Ensayar(d,i+1,solución)
      hasta_que solución o  (d[i] = n)
      si no solución entonces
         d[i] ← 0
      fin_si
   fin_si
fin_procedimiento
```



Estructuras de datos (arrays y registros)

6.1. Datos estructurados

Una estructura de datos es una colección de datos que se caracterizan por su organización y las operaciones que se definen en ella. Los datos de tipo estándar pueden ser organizados en diferentes estructuras de datos: estáticas y dinámicas.

Las estructuras de datos estáticas son aquellas en las que el espacio ocupado en memoria se define en tiempo de compilación y no puede ser modificado durante la ejecución del programa; por el contrario, en las estructuras de datos dinámicas el espacio ocupado en memoria puede ser modificado en tiempo de ejecución. Estructuras de datos estáticas son los arrays y los registros y las estructuras dinámicas son listas, árboles y grafos (estas estructuras no son soportadas en todos los lenguajes).

La elección de la estructura de datos idónea dependerá de la naturaleza del problema a resolver y, en menor medida, del lenguaje. Las estructuras de datos tienen en común que un identificador, nombre, puede representar a múltiples datos individuales.

6.2. Arrays*

Un array es una colección de datos del mismo tipo, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común. Para referirse a un determinado elemento de un array se deberá

* *En Latinoamérica, el termino inglés* array, *se suele traducir casi siempre por la palabra* arreglo.

utilizar un índice, que especifique su posición relativa en el array. Los arrays podrán ser

- Unidimensionales, también llamados Vectores
- Bidimensionales, denominados Tablas o Matrices
- Multidimensionales, con tres o más dimensiones

elemento 1
elemento 2
elemento 3
......
elemento m

Array unidimensional

elemento 1,1	......	elemento 1,n
elemento 2,1	......	elemento 2,n
elemento 3,1	......	elemento 3,n
......	......	......
elemento m,1	......	elemento m,n

Array bidimensional

elemento 1,1,2	......	*elemento 1,n,2*
elemento 1,1,1	......	*elemento 1,n,1*
elemento 1,1,1	......	elemento 1,n,1
elemento 2,1,1	......	elemento 2,n,1
elemento 3,1,1	......	elemento 3,n,1
......	......	......
elemento m,1,1	......	elemento m,n,1

Array tridimensional

Puesto que la memoria de la computadora es lineal, sea el array del tipo que sea, deberá estar linealizado para su disposición en el almacenamiento.

6.2.1. Arrays unidimensionales

Todo dato que se vaya a utilizar en un algoritmo ha de haber sido previamente declarado. Los arrays no son una excepción y lo primero que se debe hacer es crear el tipo, para luego poder declarar datos de dicho tipo.

Al ser un tipo estructurado, la declaración se hará en función de otro tipo estándar o previamente definido, al que se denominará tipo base, por ser todos los elementos constituyentes de la estructura del mismo tipo.

```
tipo
   array[<liminf>..<limsup>] de <tipo_base> : <nombre_del_tipo>
var
   <nombre_del_tipo>: <nombre_del_vector>
```

El número de elementos del vector vendrá dado por la fórmula

```
<limsup> - <liminf> + 1
```

Todos los elementos del vector podrán seleccionarse arbitrariamente y serán igualmente accesibles. Se les referenciará empleando un índice que señale su posición relativa dentro del vector. Si *<nombre_del_vector>* fuera `vect` al elemento enésimo del array se le referenciaría por `vect[n]`, siendo `n` un valor situado entre el límite inferior y el límite superior. Los elementos podrán ser procesados como cualquier otra variable que fuera de dicho tipo_base.

6.2.2. Arrays bidimensionales

También se les denomina *matrices* o *tablas*. Un array bidimensional es un vector de vectores. Es por tanto un conjunto de elementos del mismo tipo en el que el orden de los componentes es significativo y en el que se necesitan especificar dos subíndices para poder identificar a cada elemento del array.

	1	2	...	1,j
1	7,5	2,3	...	...
2	3,4	6,3	...	...
...	...	...	...	...
i				m[i,j

Su declaración se haría de la siguiente forma:

```
tipo
  array[lInf..lSup,linf..lsup] de tipo_base: nombre_del_tipo
var
  <nombre_del_tipo>: <nombre_de_la_matriz>
```

que aplicado a la matriz dibujada más arriba sería:

```
tipo
   array[1..F, 1..C] de real : matriz
  //F y C habrán sido declaradas previamente como constantes
var
  matriz: m
```

La referencia a un determinado elemento de la matriz, requiere el empleo de un primer subíndice que indique la fila y un segundo subíndice que marque la columna.

Los elementos, m[i,j], podrán ser procesados como cualquier otra variable de tipo real. El número de elementos de la matriz vendrá dado por la fórmula

```
(F - 1 + 1) * (C - 1 + 1)
```

6.2.3. Recorrido de todos los elementos del array

El recorrido de los elementos de un array se realizará utilizando estructuras repetitivas, con las que manejaremos los subíndices del array. Si se trata de un vector, bastará con una estructura repetitiva. Para el recorrido de una matriz o tabla se necesitarán dos estructuras repetitivas anidadas, una que controle las filas y otra las columnas. Además, en las matrices se consideran dos posibilidades:

Recorrido por filas

Supuestas las siguientes declaraciones

```
const
   F = <valor1>
   C = <valor2>
tipo
   array[1..F, 1..C] de real : matriz
var
   matriz: m
```

y que todos los elementos de la matriz contienen información válida, escribir el pseudocódigo para que se visualice primero el contenido de los elementos de la primera fila, a continuación el contenido de los de la segunda, etc.

```
desde i ← 1 hasta F hacer
   desde j ← 1 hasta C hacer
      escribir(m[i,j])
   fin_desde
fin_desde
```

Recorrido por columnas

Tal y como aparece a continuación, se mostrará primero el contenido de los elementos de la primera columna, luego el de los elementos de la segunda columna, etc.

```
desde j ← 1 hasta C hacer
   desde i ← 1 hasta F hacer
      escribir(m[i,j])
   fin_desde
fin_desde
```

Para recorrer los elementos de una matriz de n dimensiones, utilizaremos n estructuras repetitivas anidadas.

6.2.4. Arrays como parámetros

Los arrays podrán ser pasados como parámetros, tanto a procedimientos como a funciones. Para ello se debe declarar algún parámetro formal del mismo tipo que el array que constituye el parámetro actual. Por ejemplo:

```
algoritmo  pasodearrays
tipo
  array[1..F, 1..C] de real : arr
var
  arr: a
  entero: b
  ....................
inicio
  ....................
  b ← recibearray ( a )
  ....................
fin

entero función recibearray(E arr: m)
  //Los parámetros actuales y los formales no necesitan
  //coincidir en el nombre)
  .....
  inicio
     ........
     ........
fin_función
```

6.3. Registros

Un registro es un dato estructurado, formado por elementos lógicamente relacionados, que pueden ser del mismo o de distinto tipo, a los que se denomina *campos*. Los campos de un registro podrán ser de un tipo estándar o de otro tipo registro previamente definido.

Ejemplo de un dato de este tipo podría ser el que permitiera almacenar la situación de un punto en el plano, compuesta por dos números reales. De igual forma, si lo que se desea es describir a una persona, se podría utilizar un dato de tipo registro para almacenar, agrupada, la información más relevante.

Un tipo registro se declarará de la siguiente forma:

```
tipo
  registro: <nombre_del_tipo>
     <tipo_de_dato1>:<nombre_de_campo1> [,<nombre_de_campo2>] [...]
     <tipo_de_dato2>:<nombre_de_campoX> [,<nombre_de_campoY>] [...]
     ........
  fin
```

Para declarar una variable de tipo registro:

```
var
  <nombre_del_tipo>: <nombre_de_variable>
```

Para acceder a un determinado campo de un registro se utilizará, como suelen emplear la mayoría de los lenguajes el nombre de la variable de tipo registro unido por un punto al nombre del campo.

```
<nombre_de_variable>.<nombre_de_campo1>
```

Si los campos del registro fueran a su vez otros registros habrá que indicar

```
<nombre_de_variable>.<nombre_de_campo1>.<nombre_de_campo_de_campo1>
```

Los datos de tipo registro se pueden pasar como parámetros tanto a procedimientos como a funciones.

6.3.1. Arrays de registros y arrays paralelos

Los elementos de un array pueden ser de cualquier tipo, por tanto es posible la construcción de *arrays de registros*.

Otra forma de trabajar en arrays con información de distinto tipo y lógicamente relacionada es el uso de arrays paralelos, que, al tener el mismo número de elementos, se podrán procesar simultáneamente.

Array de registros 1

Nombre	Edad	
'Ana'	28	a[1]
'Carlos'	36	a[2]
...	...	....
'Juan'	34	a[n]

Arrays paralelos

Nombres		Edades	
'Ana'	Nombres[1]	28	Edades[1]
'Carlos'	Nombres[2]	36	Edades[2]
...	...	...	...
'Juan'	Nombres[n]	34	Edades[n]

Para acceder al campo nombre del 2º elemento del array A se escribiría a[2].nombre. Por ejemplo,

```
escribir(a[2].nombre)
```

presentaría en consola la cadena `Carlos`.

6.4. Ejercicios resueltos

6.1. *Determinar los valores de los vectores N y M después de la ejecución de las instrucciones siguientes:*

```
var
   array [1..3] de entero : M, N

1. M[1] ← 1
2. M[2] ← 2
```

```
3. M[3] ← 3
4. N[1] ← M[1] + M[2]
5. N[2] ← M[1] - M[3]
6. N[3] ← M[2] + M[3]
7. N[1] ← M[3] - M[1]
8. M[2] ← 2 * N[1] + N[2]
9. M[1] ← N[2] + M[1]
```

	M[1]	M[2]	M[3]	N[1]	N[2]	N[3]
1	1	-	-	-	-	-
2	1	2	-	-	-	-
3	1	2	3	-	-	-
4	1	2	3	3	-	-
5	1	2	3	3	-2	-
6	1	2	3	3	-2	5
7	1	2	3	2	-2	5
8	1	6	3	2	-2	5
9	1	6	3	2	-2	5

6.2. *Leer un vector V desde un terminal de entrada.*

Análisis del problema

DATOS DE ENTRADA: `V` (el array que vamos a rellenar)
DATOS AUXILIARES: `n` (número de elementos del array), `i` (contador que controla el número de lecturas y que proporciona además el índice de los elementos del array)

La lectura de un array es una operación repetitiva, por lo que se debe utilizar alguna estructura iterativa. Si es posible, lo mejor es una estructura **`para`**, ya que se conoce de antemano el número de iteraciones que se han de realizar.

Diseño del algoritmo

```
algoritmo ejercicio_6_2
const
   //el número máximo de elementos de un array lo tomamos como
```

```
  // una constante
  MáxArray = ...
var
  array [1..MáxArray] de entero : v
  entero : i, n
inicio
  leer(n)
  desde i ← 1 hasta n hacer
    leer(V[i])
  fin_desde
fin
```

6.3. *Escribir un algoritmo que permita calcular el cuadrado de los 100 primeros números enteros y a continuación escribir una tabla que contenga dichos cuadrados.*

Análisis del problema

DATOS DE SALIDA:	`cuadr` (el vector que guarda los cuadrados de los 100 primeros enteros)
DATOS AUXILIARES:	`i` (variable del bucle)

Si desea hacer la operación en dos fases, una de cálculo y otra de escritura, se ha de utilizar un array. Primero se hace un bucle en el que se vayan sacando los números del 1 al 100, se calcula su cuadrado y se asigna el resultado a un elemento del array. Un segundo bucle, que se repetirá también 100 veces nos servirá para escribir todos los elementos del array.

Diseño del algoritmo

```
algoritmo ejercicio_6_3
var
  array [1..100] de entero : cuadr
  entero : i
inicio
  //rellenar el array con los cuadrados
  desde i ← 1 hasta 100 hacer
    cuadr[i] ← i * i
  fin_desde
  //escritura del array
  desde i ← 1 hasta 100 hacer
    escribir(cuadr[i])
  fin_desde
fin
```

6.4. *Se tienen N temperaturas almacenadas en un array. Se desea calcular su media y obtener el número de temperaturas mayores o iguales que la media.*

Análisis del problema

DATOS DE SALIDA:	`media` (la media de las N temperaturas), `mayores` (contador que guardará el número de temperaturas mayores que la media)
DATOS DE ENTRADA:	`N` (número de temperaturas), `temp` (array de n elementos donde se guardan las temperaturas)
DATOS AUXILIARES:	`i` (variable de los bucles que indica también el índice de los elementos del array).

El algoritmo debe tener dos bucles. En el primero se leen por teclado todos los elementos del array. Mediante una función, se halla la media del array. Una vez leídos los elementos y hallada la media, se recorre otra vez el array para determinar cuántos elementos son superiores a la media. Cada vez que se encuentra un dato que cumpla esa condición se incrementa el contador `mayores` en una unidad.

Diseño del algoritmo

```
algoritmo ejercicio_6_4
const
  MáxArray = ...
tipos
  array[1..MáxArray] de real : vector
var
  Vector : temp
  real : mediatemp
  entero : mayores, n, i
inicio
  leer(n)
  desde i ← 1 hasta n hacer
    leer(temp[i])
  fin_desde
  mediatemp ← media(temp,n)
  mayores ← 0
  desde i ← 1 hasta n hacer
    si temp[i] >= mediatemp entonces
      mayores ← mayores + 1
    fin_si
  fin_desde
  escribir(mediatemp, mayores)
fin

entero función media(E Vector : v ;  E entero n )
var
  real : m
  entero : i
inicio
  m ← 0
```

```
  desde i ← 1 hasta n hacer
     m ← m + temp[i]
  fin_desde
  devolver(m/n)
fin_función
```

6.5. *Calcular la suma de todos los elementos de un vector de dimensión 100, así como su media aritmética.*

Análisis del problema

DATOS DE SALIDA:	suma (suma de los elementos del vector), media (media de los elementos)
DATOS DE ENTRADA:	vector (el array que contiene los elementos a sumar)
DATOS AUXILIARES:	i (variable del bucle)

Mediante un bucle **desde** se leen y suman los elementos del vector. Una vez hecho esto se calcula la media (suma / 100) y se escriben la media y la suma.

Diseño del algoritmo

```
algoritmo ejercicio_6_5
var
  array[1..100] de entero : vector
  real : media
  entero : suma, i
inicio
  media ← 0
  suma ← 0
  desde i ← 1 hasta 100 hacer
     leer(vector[i])
     suma ← suma + vector[i]
  fin_desde
  media ← suma / 100
  escribir(media, suma)
fin
```

6.6. *Calcular el número de elementos negativos, cero y positivos de un vector dado de 60 elementos.*

Análisis del problema

DATOS DE SALIDA: pos (contador de elementos positivos), cero (contador de ceros), neg

	(contador de elementos negativos)
DATOS DE ENTRADA:	`lista` (vector a analizar)
DATOS AUXILIARES:	`i` (variable del bucle)

Suponiendo que se tiene leído el array desde algún dispositivo de entrada, se ha de recorrer el array con un bucle **desde** y dentro de él comprobar si cada elemento `lista[i]` es igual, mayor o menor que 0, con lo que se incrementará el contador correspondiente.

Diseño del algoritmo

```
algoritmo ejercicio_6_6
var
  array [1..60] de entero : lista
  entero : pos, neg, cero, i
inicio
  pos ← 0
  neg ← 0
  cero ← 0
  //lectura del array
  desde i ← 1 hasta 60 hacer
    si lista[i] > 0 entonces
      pos ← pos + 1
    si_no
      si lista[i] < 0 entonces
        neg ← neg + 1
      si_no
        cero ← cero + 1
      fin_si
    fin_si
  fin_desde
  escribir(pos,neg,cero)
fin
```

6.7. Rellenar una matriz identidad de 4 por 4 elementos.

Análisis del problema

DATOS DE SALIDA:	`matriz` (matriz identidad de 4 filas y cuatro columnas)
DATOS AUXILIARES:	`i` (índice de las filas), `j` (índice de las columnas)

Una matriz identidad es aquella en la que la diagonal principal está llena de unos y el resto de los elementos son 0. Como siempre que se quiere trabajar con matrices de dos dimensiones, se han de utilizar dos bucles anidados, que en este caso serán de 1 hasta 4. Para rellenar la matriz identidad, se ha

de comprobar que los índices de los bucles -i y j. Si son iguales matriz[i,j] será igual a 1, en caso contrario se asigna 0 al elemento i,j.

Diseño del algoritmo

```
algoritmo ejercicio_6_7
var
   array [1..4,1..4] de enteros : matriz
   entero : i, j
inicio
   desde i ← 1  hasta 4 hacer
      desde j ← 1  hasta 4 hacer
         si  i = j entonces
            matriz[i,j] ← 1
         si_no
            matriz[i,j] ← 0
         fin_si
      fin_desde
   fin_desde
fin
```

6.8. *Diseñar un algoritmo que calcule el mayor valor de una lista L de N elementos.*

Análisis del problema

DATOS DE SALIDA:	N (número de elementos de la lista), máx (el elemento mayor de la lista)
DATOS DE ENTRADA:	L (lista a analizar)
DATOS AUXILIARES:	i (índice de la lista)

En primer lugar se ha de leer la dimensión de la lista (N) y rellenarla (en el ejercicio se rellena simplemente con la instrucción **LeerLista**(L, N)). Para hallar el elemento mayor de un array, se debe inicializar la variable máx con el primer elemento de la lista (L[1]) y hacer un bucle desde 2 hasta N en el que se comparará cada elemento L[i] con máx. Si L[i] es mayor que máx, L[i] será el nuevo máximo.

Diseño del algoritmo

```
algoritmo ejercicio_6_8
const
   //Se considera que el array tiene un máximo de 100 elementos
   MáxArray = 100
tipo
   array [1..MáxArray] de enteros : lista
```

```
var
   lista : L
   entero : máx,N,i
inicio
   leer(N)
   LeerLista(L, N)
   máx ← L[1]
   desde i ← 2 hasta N hacer
      si L[i] > máx entonces
         máx ← L[i]
      fin_si
   fin_desde
   escribir(máx)
fin
```

6.9. *Dada una lista L de N elementos diseñar un algoritmo que calcule de forma independiente la suma de los números pares y la suma de los números impares.*

Análisis del problema

DATOS DE SALIDA:	spar (suma de números pares), simpar (suma de los números impares)
DATOS DE ENTRADA:	L (lista a analizar)
DATOS AUXILIARES:	i (variable del bucle)

Una vez leído el array se han de analizar cada uno de sus elementos comprobando si L[i] es par o impar mediante el operador resto (es par si L[i] **mod** 2 = 0). Si esta condición se cumple se acumula el valor de L[i] en spar, en caso contrario se acumulará en simpar.

Diseño del algoritmo

```
algoritmo ejercicio_6_9
const
  MáxArray = ...
var
  array [1..MáxArray] de entero : L
  entero : N,i,spar,simpar
inicio
   .
   .
   // lectura de N elementos del array L
   .
   .
   spar ← 0
   simpar ← 0
```

```
  desde i ← 1 hasta n hacer
     si L[i] mod 2 = 0 entonces
        spar ← spar + l[i]
     si_no
        simpar ← simpar + l[i]
     fin_si
  fin_desde
  escribir(spar,simpar)
fin
```

6.10. *Escribir el pseudocódigo que permita escribir el contenido de una tabla de dos dimensiones (5 x 4).*

Análisis del problema

DATOS DE SALIDA: `tabla` (la tabla a escribir)
DATOS AUXILIARES: `i,j` (índices de las fila y columna de cada elemento del array)

Si se supone la tabla ya creada, para escribirla se han de recorrer todos sus elementos. Para recorrer un array de dos dimensiones, hay que implementar 2 bucles anidados, uno para recorrer la filas y otro para recorrer las columnas. Dentro del bucle interno, lo único que hay que hacer es escribir el elemento i,j del array.

Diseño del algoritmo

```
algoritmo ejercicio_6_10
var
  array [1..5,1..4] de entero : tabla
  entero : i, j
inicio
  desde i ← 1 hasta 5 hacer
     desde j ← 1 hasta 4 hacer
        escribir(tabla[i,j])
     fin_desde
  fin_desde
fin
```

6.11. *Se desea diseñar un algoritmo que permita obtener el mayor valor almacenado en una tabla VENTAS de dos dimensiones (4 filas y 5 columnas).*

Análisis del problema

DATOS DE SALIDA: máx (el elemento mayor de la tabla)
DATOS DE ENTRADA: VENTAS (la tabla a analizar)
DATOS AUXILIARES: i,j (índices de la fila y columna de cada elemento), fmáx, cmáx (fila y columna del elemento mayor de la lista)

En este ejercicio se empleará otro método para hallar el máximo. En vez de hacer la búsqueda tomando como referencia el contenido del elemento mayor, se usa como referencia la posición (fila y columna) que ocupa dicho elemento.

Después de leer la tabla, se inicializan los valores de fmáx y cmáx (fila y columna que ocupa el elemento mayor) a 1. Se debe recorrer el array con dos bucles anidados. En el bucle interno se compara VENTAS[i,j], con VENTAS[fmáx,cmáx]. Si es mayor, fmáx y cmáx pasarán a tener el valor que en esos momentos contengan i y j. Al final del bucle en fmáx y cmáx estará guardada la posición del elemento mayor por lo que asignamos a la variable máx el contenido de VENTAS[fmáx,cmáx] y lo escribimos.

Diseño del algoritmo

```
algoritmo ejercicio_6_11
var
  array [1..4,1..5] de entero : VENTAS
  entero : i,j,fmáx,cmáx,máx
inicio
  //lectura del array VENTAS
  fmáx ← 1
  cmáx ← 1
  desde i ← 1  hasta 4 hacer
    desde j ← 1  hasta 5 hacer
      si VENTAS[i,j] > VENTAS[fmáx,cmáx] entonces
        fmáx ← i
        cmáx ← j
      fin_si
    fin_desde
  fin_desde
  máx ← VENTAS[fmáx,cmáx]
  escribir(máx)
fin
```

6.12. Hacer diferentes listados de una lista de 10 números según el siguiente criterio:

si número >=0 y número < 50, ponerlo en LISTA1
si número >=50 y número <100, ponerlo en LISTA2
si número >=100 y número <=150, ponerlo en LISTA3

Análisis del problema

DATOS DE SALIDA: LISTA1, LISTA2, LISTA3
DATOS DE ENTRADA: LISTA (lista que contiene los números iniciales)
DATOS AUXILIARES: i (índice de las lista original), conta1, conta2, conta3 (índices de cada una de las listas de salida)

Se han de utilizar cuatro arrays que tendrán todos la dimensión del array original LISTA (10 elementos). Una vez leido el array LISTA, se debe recorrer para comprobar cada uno de sus elementos e introducirlo en la lista correspondiente incrementando también su índice.

Una vez llenas las tres listas de salida, se escriben mediante tres bucles que irán cada uno desde 1 hasta el último valor del índice correspondiente. Como la operación de escritura va a ser igual para las tres listas se implementará un procedimiento que se encargue de la operación de lectura. En dicho procedimiento se pasa la lista y el número de elementos de la misma (conta1, conta2, conta3). Al utilizar un procedimiento con un tipo de datos estructurado, es necesario crear un tipo de datos definido por el usuario (vector).

Diseño del algoritmo

```
algoritmo ejercicio_6_12
tipo
   array [1..10] de entero : vector
var
   vector  : LISTA,LISTA1,LISTA2,LISTA3
   entero  : i,conta1,conta2,conta3
inicio
   conta1 ← 0
   conta2 ← 0
   conta3 ← 0
   //leemos el array LISTA
   desde i ← 1 hasta 10 hacer
      si LISTA[i] > 0 entonces
         si LISTA[i] < 50 entonces
            conta1 ← conta1 + 1
            LISTA1[conta1] ← LISTA[i]
         si_no
            si LISTA[i] < 100 entonces
               conta2 ← conta2 + 1
               LISTA2[conta2] ← LISTA[i]
            si_no
               si LISTA[i] <= 150 entonces
                  conta3 ← conta3 + 1
                  LISTA3[conta3] ← LISTA[i]
               fin_si
            fin_si
         fin_si
```

```
      fin_si
   fin_desde
   EscribirVector(LISTA1,conta1)
   EscribirVector(LISTA2,conta2)
   EscribirVector(LISTA3,conta3)
fin

procedimiento EscribirVector(E vector: v ;E entero : n)
var
   entero : i
inicio
   desde i ← 1 hasta n hacer
      escribir(v[i])
   fin_desde
fin_procedimiento
```

6.13. *Rellenar un vector a de N elementos con enteros consecutivos de forma que para cada elemento* $A_i \leftarrow i$.

Análisis del problema

DATOS DE ENTRADA: A (el array a rellenar), N (número de elementos del array)
DATOS AUXILIARES: i (índice del array)

Se debe ejecutar un bucle **desde** N veces. Dentro del bucle hay que hacer la asignación A[i] ← i.

Diseño del algoritmo

```
algoritmo Ejercicio_6_13
const
  MáxArray = ...
var
  entero : i
  array [1..MáxArray] de entero  : A
inicio
  desde i ← 1 hasta MáxArray hacer
    A[i] ← i
  fin_desde
fin
```

6.14. *Escribir un programa para introducir una serie de números desde el teclado. Utilizar un valor centinela -1E5 para terminar la serie. El programa calculará e imprimirá el número de valores leídos, la suma y la media de la tabla. Además, generará una tabla de dos dimensiones en el*

que la primera columna será el propio número y la segunda indicará cuánto se desvía de la media.

Análisis del problema

DATOS DE SALIDA:	`conta` (número de valores leidos), `suma` (suma de todos los valores), `media` (media de dichos valores), `desviación` (tabla con las desviaciones de cada elemento con respecto a la media).
DATOS DE ENTRADA:	`vector` (el vector que leemos), `núm` (cada uno de los números que leemos)
DATOS AUXILIARES:	`i` (índice del vector)

Se debe dimensionar el array a un número lo suficientemente grande como para que quepan los valores a procesar. Como ese número no está determinado, se dimensiona al número máximo de valores previsto (representado por `MáxLista`).

El bucle utilizado para procesar el vector no puede ser un bucle **desde**, sino que se debe utilizar uno **repetir** o un **mientras**. Dentro de este bucle de lectura, se lee el elemento, se acumula su valor en suma y se incrementa el índice i. La variable leída será núm, y una vez verificado que es distinta de -1E5, se asigna el valor `vector[i]`.

El número de valores procesados (`conta`) será igual al último valor de `i`. Se halla la media (`suma / conta`) y se escribe `conta`, `media` y `suma`.

Para hallar la desviación de cada uno de los elementos se hará otro bucle (ahora sirve un bucle **desde**) en el que el índice vaya tomando valores entre 1 y `conta`. Dentro del bucle se calcula la desviación de elemento y escribimos ambos.

Diseño del algoritmo

```
algoritmo  ejercicio_6_14
const
   MáxArray = ...
var
   array[1..MáxArray] de real  : vector
   array[1..MáxArray,1..2] de real  : desviación
   entero : i,j,conta
   real  : núm,suma,media,desviación
inicio
   i ← 0
   suma ← 0
   leer(núm)
   mientras núm <> -1E5 hacer
      i ← i + 1
      vector[i] ← núm
      suma ← suma + vector [i]
      leer(núm)
```

```
   fin_mientras
   conta ← i
   media ← suma /conta
   escribir(conta,suma,media)
   desde i ← 1 hasta conta hacer
      desviación[i, 1] ← vector[i]
      desviación[i, 2] ← vector[i] - media
   fin_desde
fin
```

6.15. Dado un vector X compuesto por N elementos, se desea diseñar un algoritmo que calcule la desviación estándar D.

$$D = \frac{\sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2}}{n - 1}$$

$$\overline{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Análisis del problema

DATOS DE SALIDA: `DesviaciónEst` (función que calcula la desviación estandar)
DATOS DE ENTRADA: `X` (el vector), `N` (número de elementos)
DATOS AUXILIARES: `i` (índice del array), `media` (media de los elementos)

Se desarrolla una función que permita calcular la desviación estándar de `N` elementos de una lista de números reales. La función va a recibir como parámetros de entrada el vector `X` –al que damos un tipo de datos vector que debe estar definido previamente– y el número de elementos. Una función se encargará de hallar la media. Dentro de esta función, es necesario saber la suma de los elementos, para lo que utilizaremos otra función `SumaReal`. Una vez calculada la media, mediante un bucle se halla la suma de los cuadrados de cada elemento menos la media, necesaria para poder hallar la desviación. Por último se calcula la desviación estándar por la fórmula dada anteriormente.

Diseño del algoritmo

```
real función DesviaciónEst(E vector : x ; E entero : n)
var
   real : media, suma
   entero : i
inicio
   media ← MediaReal(X,n)
   suma ← 0
```

```
   desde i ← 1 hasta n hacer
      suma ← suma + cuadrado(X[i] - media)
   fin_desde
   devolver(raíz2(suma) / (n-1))
fin_función

real función MediaReal( E vector : v ;  E entero : n)
inicio
   devolver(SumaReal( v,n) / n)
fin_función

real función SumaReal( E vector : v ; E entero : n)
var
   real : suma
   entero : i
inicio
   suma ← 0
   desde i ← 1 hasta n hacer
      suma ← suma + v[i]
   fin_desde
   SumaReal ← suma
fin_función
```

6.16. Leer una matriz de 3 x *3.*

Análisis del problema

DATOS DE ENTRADA: `matriz` (la tabla o matriz de 3 x 3)
DATOS AUXILIARES: `i,j` (índices de las fila y columna de cada elemento del array)

Para leer una matriz es necesario utilizar dos bucles anidados. Dentro del bucle interior se incluye una operación de lectura con cada elemento de la matriz.

Diseño del algoritmo

```
algoritmo ejercicio_6_16
var
   array [1..3,1..3] enteros  : matriz
   entero : i, j
inicio
   desde i ← 1 hasta 3 hacer
      desde j ← 1 hasta 3 hacer
         leer(matriz[i,j])
      fin_desde
   fin_desde
```

```
fin
```

6.17. Escribir el algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T de n filas y m columnas.

Análisis del problema

DATOS DE SALIDA:	`pos` (suma de los elementos positivos), `neg` (suma de los elementos negativos)
DATOS DE ENTRADA:	`T` (tabla a procesar), `n` (número de filas), `m` (número de columnas)
DATOS AUXILIARES:	`i,j` (índices de los elementos de la tabla)

Después de leer la tabla, se recorre mediante dos bucles anidados. En el bucle interno se ha de comprobar si `T[i,j]` es positivo o negativo. Si es mayor que 0 se acumula el valor de `T[i,j]` en `pos`. Si es menor que 0, se hace la misma operación en `neg`. En otro caso, `T[i,j]` sería igual a 0, por lo que no se ha de hacer nada.

Diseño del algoritmo

```
algoritmo ejercicio__6_17
const
  MáxFila = ...
  MáxCol = ...
var
  array [1..MáxFila,1..MáxCol] de entero : T
  entero : T
inicio
  leer(n,m)
  // lectura de la tabla
  pos ← 0
  neg ← 0
  desde i ← 1 hasta n hacer
    desde j ← 1 hasta m hacer
      si T[i,j] > 0 entonces
        pos ← pos + T[i,j]
      si_no
        si T[i,j] < 0 entonces
          neg ← neg + T[i,j]
        fin_si
      fin_si
    fin_desde
  fin_desde
  escribir(pos,neg)
fin
```

6.18. Supongamos que existen N ciudades en la red ferroviaria de un país, y que sus nombres están almacenados en un vector llamado CIUDAD. Diseñar un algoritmo en el que se lea el nombre de cada una de las ciudades y los nombres con los que está enlazada.

Análisis del problema

DATOS DE SALIDA:	Listado de ciudades
DATOS DE ENTRADA:	`CIUDAD` (vector donde se guardan los nombres de las ciudades), `ENLACE` (array donde representamos los enlaces de unas ciudades con otras), N (número de ciudades)
DATOS AUXILIARES:	`i,j` *(índices de los arrays)*

En primer lugar, hay que ver las ciudades que están enlazadas entre sí; para ello se utiliza una tabla de N x N elementos que representan las ciudades. Por ejemplo, la indicación de que la ciudad 3 está enlazada con la 5 (`CIUDAD[3,5]`) se realiza mediante una marca, por ejemplo un 1. Si no están enlazadas en dicho elemento se pone un 0. La tabla de ciudades, será un array triangular: sólo se ha de llenar medio array, ya que si la ciudad 1 está enlazada con la ciudad 2, la ciudad 2 lo estará con la 1. Por lo tanto el bucle interno irá desde *i+1* hasta *n*.

Para sacar los nombres de las ciudades se necesita disponer de otro array en el que el elemento 1 contenga el nombre de la ciudad 1, el 2 el de la 2 y así hasta el número de ciudades, es decir, hasta N.

Después de llenar el array de enlaces y el de ciudades (para hacer el listado que pide el problema) es necesario recorrer toda la tabla de enlaces mediante 2 bucles anidados. En el bucle externo se debe escribir el nombre de la ciudad a la que se sacan los enlaces; es decir si se está en la ciudad i, se debe escribir `CIUDAD[i]`. En el bucle interno se debe comprobar el contenido de `ENLACE[i,j]`. Si éste es distinto de 0, se escribe la ciudad con la que está enlazada (`CIUDAD[j]`).

Diseño del algoritmo

```
algoritmo ejercicio_6_18
const
   n = ...
var
   array[1..n] de cadena : ciudad
   array[1..n,1..n] de entero : n
   entero : i, j
inicio
   // lectura del array de ciudades y de enlaces
   desde i ← 1 hasta n hacer
      leer(ciudad[i])
      desde j ← i+1 hasta n hacer
         leer(enlace[i,j])
      fin_desde
   fin_desde
   // listado de las ciudades con sus enlaces
```

```
   desde i ← 1 hasta n hacer
      escribir(ciudad[i])
      desde j ← 1 hasta n hacer
         si enlace[i,j] = 1 entonces
            escribir(ciudad[j])
         fin_si
      fin_desde
   fin_desde
fin_desde
```

6.19. Determinar si una matriz de tres filas y tres columnas es un cuadrado mágico. Se considera un cuadrado mágico *aquel en el cual las filas, columnas, y las diagonales principal y secundaria suman lo mismo.*

Análisis del problema

DATOS DE SALIDA:	Mensaje queindica si es un cuadrado mágico
DATOS DE ENTRADA:	`cuad` (la matriz que vamos a comprobar)
DATOS AUXILIARES:	`i,j` (índices de los arrays), `mágico` (variable lógica que valdrá verdadero si la matriz es un cuadrado mágico),`suma` (sumador que acumula la suma de los elementos de la primera fila), `diagonal1` y `diagonal2` (suma de los valores de la diagonal principal y secundaria respectivamente)

Después de sumar la primera fila mediante una función `SumaFila`, se utiliz un bucle **desde**, que comprueba las siguientes filas llamando a su vez a la función `SumaFila`. Para no tener que recorrer todo el array, el bucle externo será un bucle que se repita mientras que el índice sea menor o igual que 3 o la variable `mágico` sea cierta (previamente se inicializa la variable `mágico` a **falso**). Dentro de ese bucle se ha de pasar a la función la fila que se desea sumar (`i`), y se comprueba si el valor que retorna la función es igual al valor de la suma de la primera fila (`suma`).

De forma similar se procede con la suma de las columnas en la que utilizaremos la función `SumaCol`. Para las diagonales se utiliza un bucle **desde** en el que se acumulan todos los valores de `cuad[i,i]` en la variable `diagonal1` y los valores de `cuad[i,4-i]` en `diagonal2`. Si después de esto el valor de todas las sumas de filas, columnas y diagnonales son iguales, `mágico` será verdadero, y aparecerá el mensaje diciendo que la matriz es un cuadrado mágico.

Diseño del algoritmo

```
algoritmo ejercicio_6_19
tipos
   array[1..3,1..3] de entero : tabla
var
   tabla : cuad
   entero  : suma,diagonal1,diagonal2,i
```

```
  lógico : mágico
inicio
  mágico ← V
  // lectura de la matriz
  leer(cuad)
  //hallamos la suma de la primera fila
  suma ← SumaFila(cuad,3,1)
  //hallamos la suma de las restantes filas
  i ← 1
  mientras i < 3 y mágico hacer
    i ← i + 1
    mágico ← suma = SumaFila(cuad,3,i)
  fin_mientras
  //hallamos la suma de las columnas
  i ← 0
  mientras i < 3 y mágico hacer
    i ← i + 1
    mágico ← suma = SumaCol(cuad,3,i)
  fin_mientras
  //hallamos la suma de las diagonales
  si mágico entonces  //si la variable mágico es cierta
    diagonal1 ← 0
    diagonal2 ← 0
    desde i ← 1 hasta 3 hacer
      diagonal1 ← diagonal1 + cuad[i,i]
      diagonal2 ← diagonal2 + cuad[i,4-i]
    fin_desde
  fin_si
  mágico ← mágico y (diagonal1 = suma) y (diagonal2 = suma)
  si mágico entonces
    escribir('La matriz es un cuadrado mágico')
  fin_si
fin

entero función SumaFila( E tabla : t ; E entero : n, i)
var
  entero : j, suma
inicio
  suma ← 0
  desde j ← 1 hasta n hacer
    suma ← suma + t[i,j]
  fin_desde
  devolver(suma)
fin_función

entero función SumaCol( E tabla : t ; E entero n, j)
var
  entero : i, suma
inicio
  suma ← 0
```

```
   desde i ← 1 hasta n hacer
      suma ← suma + t[i,j]
   fin_desde
   devolver(suma)
fin_función
```

6.20. Visualizar la matriz traspuesta de una matriz M de 6 x 7 elementos.

Análisis del problema

DATOS DE SALIDA:	MT (matriz traspuesta de M)
DATOS DE ENTRADA:	M (matriz original)
DATOS AUXILIARES:	i,j (índice de las matrices)

Una matriz traspuesta (MT) a otra es aquella que tiene intercambiadas las filas y las columnas. Si una matriz M tiene 6 filas y 7 columnas, MT tendrá 7 filas y 6 columnas. Mientras se lee la matriz M, se puede obtener MT, ya que a cualquier elemento M[i,j], le corresponderá MT[j,i] en la matriz MT. Una vez leída, se escibe mediante otros dos bucles **desde** anidados.

Diseño del algoritmo

```
algoritmo ejercicio_6_20
var
   array[1..6,1..7] de entero : M
   array[1..7,1..6] de entero : MT
   entero : i,j
inicio
   desde i ← 1 hasta 6 hacer
      desde j ← 1 hasta 7 hacer
         leer(M[i,j])
         MT[j,i] ← M[i,j]
      fin_desde
   fin_desde
   desde i ← 1 hasta 7 hacer
      desde j ← 1 hasta 6 hacer
         escribir(MT[i,j])
      fin_desde
   fin_desde
fin
```

6.21. Diseñar una función con la que dadas dos matrices pasadas como parámetros, comprobar si son idénticas.

Análisis del algoritmo

DATOS DE SALIDA:	`Mensaje`
DATOS DE ENTRADA:	`A`,`B` (las dos matrices) `M`,`N` (dimensiones de la matriz `A`), `O`,`P` (dimensiones de la matriz `B`)
DATOS AUXILIARES:	`i`,`j` (índices de las matrices), `idénticas` (variable lógica que valdrá verdadero si las dos matrices son iguales)

Para comprobar si dos matrices A y B son idénticas, lo primero que se ha de hacer es comprobar si sus dimensiones son iguales. Si esto es cierto se pasa a recorrer ambas matrices mediante dos bucles anidados. Como es posible que no sean idénticas, a veces no será necesario recorrer toda la matriz, sino sólo hasta que se encuentre un elemento diferente. Por esta razón no se utilizan bucles **desde** sino bucles **mientras** que se ejecutarán mientras i o j sean menores que los límites y sean idénticas las matrices (`idéntica` sea verdadero). Dentro del bucle interno se comprueba si `A[i,j]` es distinto a `B[i,j]`, en cuyo caso se pone la variable `idéntica` a falso (antes de comenzar los bucles se ha puesto a cierto la variable). Si después de recorrer las tablas, `idéntica` es verdadero, la función devolverá un valor lógico verdadero.

Diseño del algoritmo

```
// suponemos creado el tipo vector
entero función MatrizIdéntica(E vector :A,B ; E entero :M,N,O,P )
var
   entero : i,j
   lógico : idéntica
inicio
   idéntica ← M = O y N = P
   si idéntica entonces
      i ← 0
      mientras i < M y idéntica hacer
         j ← 0
         i ← i + 1
         mientras j < N y idéntica hacer
            j ← j + 1
            idéntica ← A[i,j] = B[i,j]
         fin_mientras
      fin_mientras
   fin_si
   devolver(idéntica)
fin_función
```

6.22. Un procedimiento que obtenga la matriz suma de dos matrices.

Análisis del problema

DATOS DE SALIDA: S (matriz suma)
DATOS DE ENTRADA: A, B (matrices a sumar), M, N (dimensiones de la matriz A), O, P (dimensiones de la matriz B)
DATOS AUXILIARES: i,j (índices de las matrices)

Para realizar la suma de dos matrices, es necesario que ambas tengan las mismas dimensiones, por lo que lo primero que se ha de hacer con los parámetros M, N, O y P es comprobar que M es igual a O y N es igual a P. Se leen las matrices A y B desde algún dispositivo de entrada y se realiza la suma. Si esto no ocurre, el parámetro de salida `error` devolverá un valor verdadero.

En la suma de matrices, cada elemento `S[i,j]` es igual a `A[i,j]+B[i,j]` y se debe, por lo tanto, recorrer las matrices A y B con dos bucles anidados para hallar la matriz suma.

Diseño del algoritmo

```
procedimiento MatrizSuma( E tabla : A,B ; E entero : M,N,O,P ;
                          S tabla : Suma  ; S lógico : error)
var
  entero : i, j
inicio
  si M <> O o N <> P entonces
    error ← V
  si_no
    error ← F
    desde i ← 1 hasta M hacer
      desde j ← 1 hasta O hacer
        Suma[i,j] ← A[i,j] + B[i,j]
      fin_desde
    fin_desde
  fin_si
fin_procedimiento
```

6.23. *Escribir el algoritmo de un subprograma que obtenga la matriz producto de dos matrices pasadas como parámetros.*

Análisis del problema

DATOS DE SALIDA: Prod (matriz producto), error (da verdadero si hay un error en las dimensiones)
DATOS DE ENTRADA: A, B (matrices a multiplicar), M, N (dimensiones de la matriz A), O, P (dimensiones de la matriz B)

DATOS AUXILIARES: i, j, k (índices de las matrices)

Se recorren la matrices con dos bucles **desde** anidados siempre que N sea igual a O. La matriz producto tendrá una dimensión de M x P. Cada elemento de la matriz producto Prod[i,j] será:

A[i,1]*B[1,j]+A[i,2]*B[2,j]+...+A[i,N]*B[N,j]

por lo que dentro del bucle interno se necesita otro bucle **desde** para que vaya sacando números correlativos entre 1 y N.

El subprograma que se ha de utilizar será un procedimiento, ya que aunque devuelve un dato (la matriz producto), éste es estructurado, por lo que no puede relacionarse con el valor que devuelve una función. A dicho procedimiento se pasan como parámetros de entrada las dos matrices y sus dimensiones; como parámetro de salida la matriz producto Prod y error, que será un valor lógico verdadero si las matrices no se pueden multiplicar.

Diseño del algoritmo

```
procedimiento MatrizProducto(E tabla : A,B ; E entero : M,N,O,P ;
                             S tabla : Prod ; S lógico error)

var
   entero : i,j,k
inicio
   si N <> O entonces
      error ← V
   si_no
      error ← F
      desde i ← 1 hasta M hacer
         desde j ← 1 hasta P hacer
            Prod[i,j] ← 0
            desde k ← 1 hasta N hacer
               Prod[i,j] ← Prod[i,j]+A[i,k]*B[k,j]
            fin_desde
         fin_desde
      fin_desde
   fin_si
fin_procedimiento
```

6.24. *Se tiene una matriz bidimensional de m x n elementos que se lee desde el dispositivo de entrada. Se desea calcular la suma de sus elementos mediante una función.*

Análisis del problema

DATOS DE SALIDA: suma (suma de los elementos de la matriz)

DATOS DE ENTRADA: tabla (matriz a procesar), m, n (dimensiones de tabla)
DATOS AUXILIARES: i, j (índices de la matriz)

Para obtener la suma de los elementos de una matriz, se recorren mediante dos bucles **desde** anidados. Dentro del bucle interno se hace la asignación suma ← suma + tabla[i,j]. Antes de entrar en los bucles se ha de haber inicializado la variable suma a 0.

Diseño del algoritmo

```
entero función SumaTabla( E tabla : t ; E entero : m,n)
var
   entero   : i,j,suma
inicio
   suma ← 0
   desde i ← 1 hasta m hacer
      desde j ← 1 hasta m hacer
         suma ← suma + t[i,j]
      fin_desde
   fin_desde
   devolver(suma)
fin_función
```

6.25. *Una matriz A de m filas y n columnas es simétrica si m es igual a n y se cumple que*

$$A_{i,j} = A_{j,i} \; para \; 1 < i < m \; y \; 1 < j < n$$

Se desea una función que tome como parámetro de entrada una matriz y sus dimensiones y devuelva un valor lógico que determine si se trata de una matriz simétrica o no.

Análisis del problema

DATOS DE SALIDA: Mensaje que nos indica si es o no simétrica
DATOS DE ENTRADA: A (la matriz a comprobar), n, m (dimensiones de la matriz)
DATOS AUXILIARES: i, j (índices de la matriz), simétrica (variable lógica que valdrá verdadero si A es simétrica).

Para comprobar si es simétrica primero se debe comprobar si m es igual a n y luego recorrer el array con dos bucles anidados que deberán terminar cuando se acaben de comprobar los elementos o cuando encuentren un elemento A[i,j] distinto del A[j,i]. Por lo tanto se utilizan bucles **mientras** en vez de bucles **desde**.

Además, no es preciso recorrer todo el array. Si se comprueba que A[3,4] es igual que A[4,3], no hace falta ver si A[4,3] es igual que A[3,4]. Lógicamente, cuando i es igual a j tampoco hace falta comprobar si A[3,3] es igual a A[3,3]. Por lo tanto en el bucle externo, la i deberá ir tomando

valores entre 1 y `M-1`. En el interno la `j` deberá ir tomando valores entre `i+1` y `N`. Dentro del bucle interno, si `A[i,j] <> A[j,i]` se pone la variable `simética` a falso con lo que se sale de los dos bucles.

Diseño del algoritmo

```
lógico función EsMatrizSimética(E tabla : tabla ; E entero m,n )
var
   entero : i,j
   lógico  : simética
inicio
   simética ← m = n
   si simética entonces
      i ← 0
      mientras i < M-1 y simética hacer
         i ← i + 1
         j ← i
         mientras j < N y simética hacer
            j ← j + 1
            simética  t[i,j] = t[j,i]
         fin_mientras
      fin_mientras
   fin_si
   devolver(simética)
fin_función
```

6.26. *Una empresa de venta de productos por correo desea realizar una estadística de las ventas realizadas de cada uno de los productos a lo largo del año. Distribuye un total de 100 productos, por lo que las ventas se pueden almacenar en una tabla de 100 filas y 12 columnas. Se desea conocer:*

El total de ventas de cada uno de los productos
El total de ventas de cada mes
El producto más vendido en cada mes
El nombre, el mes y la cantidad del producto más vendido

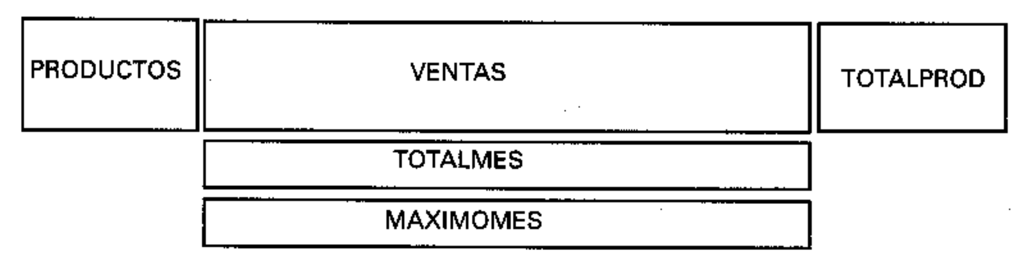
Como resultado final, se desea realizar un listado con el siguiente formato:

	Enero	Febrero	...	Diciembre	Total producto
Producto 1					
Producto 2					
...					
Producto 100					
Total mes					
Producto más vendido					

Nombre del producto y mes del producto más vendido en cualquier mes del año:

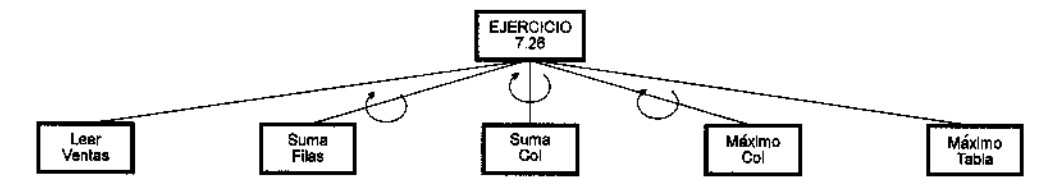
Análisis del problema

Para realizar este problema se utilizarán cinco arrays:



El array `PRODUCTOS` será un array de cadena, de 100 elementos en el que se guardarán los nombres de los productos. `VENTAS` es un array de dos dimensiones de tipo entero de 100 filas y 12 columnas que almacena las ventas de cada producto en cada mes del año. `TOTALPROD` será un array de 100 elementos en el que guarda el total de ventas anual de cada producto. `TOTALMES` tiene 12 posiciones y guarda el total de ventas de cada uno de los meses y `MÁXIMOMES` el número del producto más vendido cada uno de los meses del año.

El diseño modular del programa quedaría de la siguiente forma:



El programa principal llamará a un procedimiento que leerá los datos, es decir los nombres de los productos y las ventas de cada producto en cada uno de los meses. A partir de ahí, un bucle se encargará de ir sumando las filas, utilizando la función `SumaFilas` que ya desarrollamos anteriormente, y otro

las columnas con la función SúmaCol. En ambos bucles se almacenan los resultados en los arrays TOTALPROD y TOTALMES respectivamente. Una cuarta función se encargará de buscar la posición del máximo elemento de cada columna y una última buscará la fila y la columna (fmáx y cmáx)de elemento mayor del array para poder obtener el nombre, el mes y la cantidad del producto más vendido.

Para terminar, el listado se realizará dentro del propio programa principal que utilizará los arrays anteriores para la presentación de los resultados.

Diseño del algoritmo

```
algoritmo ejercicio_6_26
tipo
   array[1..100,1..12] de entero : tabla
   array[1..12] de entero : vector
   array[1..100] de cadena : productos
   array[1..100] de entero : totales
var
   tabla : tabla
   vector : TotalMes, Máximos
   producto : prod
   totales : tot
   entero : i,fmáx,cmáx
inicio
   LeerDatos(ventas,prod,100,12)
   // cálculo del array con los totales por producto
   // (suma de filas)
   desde i ← 1 hasta 100 hacer
      tot[i] ← SumaFila(ventas,12,i)
   fin_desde
   // cálculo del array con los totales por mes
   // (suma de columnas)
   desde j ← 1 hasta 12 hacer
      TotalMes[j] ← SumaCol(ventas,100,j)
   fin_desde
   // cálculo del array con los máximos por mes
   //(máximo de columna)
   desde j ← 1 hasta 12 hacer
      Máximos[j] ← MáximoCol(ventas,100,j)
   fin_desde
   // obtención de la fila y columna de la tabla donde
   // se encuentra el máximo
   MáximoTabla(ventas,100,12,fmáx,cmáx)
   // listado de resultados
   desde i ← 1 hasta 100 hacer
      escribir(prod[i])
      desde j ← 1 hasta 12 hacer
         escribir(ventas[i,j])  // escribir en la
                     // misma línea
```

```
      fin_desde
      // salto de línea
   fin_desde
   // escribir la fila con los totales por mes
   desde j ← 1 hasta 12 hacer
      escribir(TotalMes[j])  // escribir en la
                // misma línea
   fin_desde
   // escribir la fila con los máximos por mes
   // Como deseamos escribir el nombre del producto, y el
   // array está lleno con
   // la fila que ocupa la posición del máximo,
   // debemos escribir prod[Máximos[j]]
   desde j ← 1 hasta 12 hacer
      escribir(Prod[Máximos[j]])  // escribir en la
                  // misma línea
   fin_desde
   // escribir el producto,el número del mes y la cantidad
   // que más se ha vendido
   escribir(prod[fmáx],cmáx,ventas[fmáx,cmáx])
fin

procedimiento LeerDatos( S tabla : v : tabla  ; S producto : p ;
                         E entero : m,n)
var
   entero : i, j
inicio
   desde i ← 1 hasta m hacer
      leer(prod[i])
      desde j ← 1 hasta n hacer
         leer(ventas[i,j])
      fin_desde
   fin_desde
fin_procedimiento

// los procedimientos SumaFila y SumaCol ya se han implementado
// anteriormente

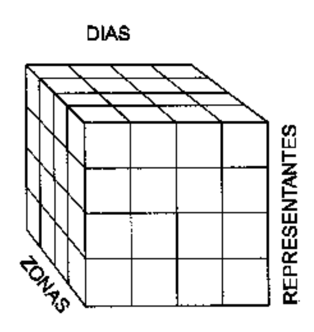
entero función MáximoCol( E tabla : t ; E entero : n,j)
var
   entero : máx, i
inicio
   máx ← 1
   desde i ← 2 hasta n hacer
      si t[i,j] > t[máx,j] entonces
         máx ← i
      fin_si
   fin_desde
   devolver(máx)
fin_función
```

```
procedimiento MáximoTabla( E tabla : v  ; E entero : m,n ;S entero : f,c)
var
  entero : i, j
inicio
  f ← 1
  c ← 1
  desde i ← 1  hasta m hacer
    desde j ← 1  hasta n hacer
      si t[i,j] > t[f,c] entonces
        f ← i
        c ← j
      fin_si
    fin_desde
  fin_desde
fin_procedimiento
```

6.27. *Una fábrica de muebles tiene 16 representantes que viajan por toda España ofreciendo sus productos. Para tareas admnistrativas el pais está dividido en cinco zonas: Norte, Sur, Este, Oeste y Centro. Mensualmente almacena sus datos y obtiene distintas estadísticas sobre el comportamiento de sus representantes en cada zona. Se desea hacer un programa que lea los datos de todos los representantes con sus ventas en cada zona y calcule el total de ventas de una zona introducida por teclado, el total de ventas de un vendedor introducido por teclado en cada una de las zonas y el total de ventas de un día y para cada uno de los representantes.*

Análisis del Problema

Una de las formas posibles de almacenar estos datos sería un array de 3 dimensiones. Se puede considerar que las filas son los representantes –de 1 a 16–, las columnas los días del mes –si se consideran meses de 31 días, de 1 a 31– y que esta estructura se repite cinco veces, una vez por cada zona. De esta forma se podría representar el array de la siguiente forma:



Al trabajar con un array de tres dimensiones es preciso utilizar tres índices, `r` para los representantes, `d` para los días y `z` para las zonas.

Una vez leído el array, para el primer proceso, se ha de introducir el número de la zona (`númzona`)

y con esa zona fija, recorrer las filas y las columnas, acumulando el total en un acumulador. Para obtener el total de ventas de un vendedor, se introduce el número de representante (`númrep`) y con dicho índice fijo recorrer la tabla por zonas y días. Por fin, para el último punto se introduce el número del día (`númdía`) y recorreremos la tabla por representantes y zonas.

Diseño del algoritmo

```
algoritmo ejercicio_6_27
var
   array[1..16,1..31,1..5] de entero : ventas
   entero    : r,d,z,númzona,númrep,númdía,suma
inicio
   // lectura de la tabla
   desde r ← 1 hasta 16 hacer
      desde d ← 1 hasta 31 hacer
         desde z ← 1 hasta 5 hacer
            leer(ventas[r,d,z])
         fin_desde
      fin_desde
   fin_desde
   // cálculo del total de una zona
   leer(númzona)
   suma ← 0
   desde d ← 1 hasta 31 hacer
      desde r ← 1 hasta 16 hacer
            suma ← suma + ventas[r,d,númzona]
      fin_desde
   fin_desde
   escribir(suma)
   // cálculo de las ventas de un representante para cada una de las zonas
   leer(númrep)
   suma ← 0
   desde z ← 1 hasta 5 hacer
      suma ← 0
      desde d ← 1 hasta 31 hacer
            suma ← suma + ventas[númrep,d,z]
      fin_desde
      escribir(suma)
   fin_desde
   // cálculo de las ventas de todos los representantes
   // para un día determinado
   leer(númdía)
   suma ← 0
   desde r ← 1 hasta 16 hacer
      suma ← 0
      desde z ← 1 hasta 5 hacer
            suma ← suma + ventas[r,númdía,z]
      fin_desde
```

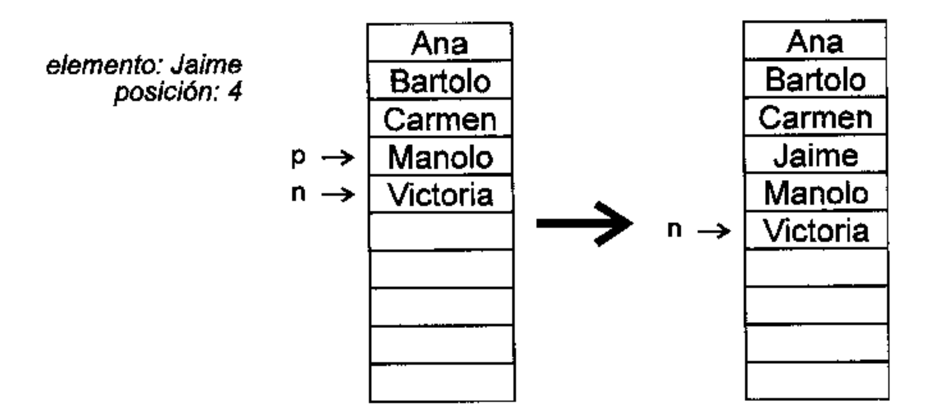
```
      escribir(suma)
   fin_desde
fin
```

6.28. *Disponemos de un array unidimensional de MáxLista elementos de tipo cadena. Se desea hacer un procedimiento que inserte un elemento dentro del array en una posición determinada respetando el orden que tenía, siempre y cuando haya sitio para hacer la operación.*

Análisis del problema

Para insertar un elemento en un array es necesario desplazar los elementos anteriores una posición hacia abajo, siempre y cuando haya sitio para insertarlo. Para controlar el número de elementos ocupados se utiliza una variable `n`. Por lo tanto sólo podremos hacer la operación si `n < MáxLista`.

Para dejar espacio para insertar el elemento se hará un bucle desde `n` hasta la posición donde se desea insertar menos uno, y hacer que cada elemento pase a la posición siguiente, es decir, a `lista[i+1]` se le asigna `lista[i]`.



Una vez hecho esto en la posición deseada se introduce el nuevo elemento y se incrementa en 1 el número de posiciones ocupadas.

Al procedimiento se le pasan como parámetros el array (`lista`), el número de elementos ocupados (`n`), una variable lógica que informa si la operación se ha realizado con éxito (`error`), la posición donde se desea insertar el elemento (`p`), y el elemento a insertar (`e`).

Diseño del algoritmo

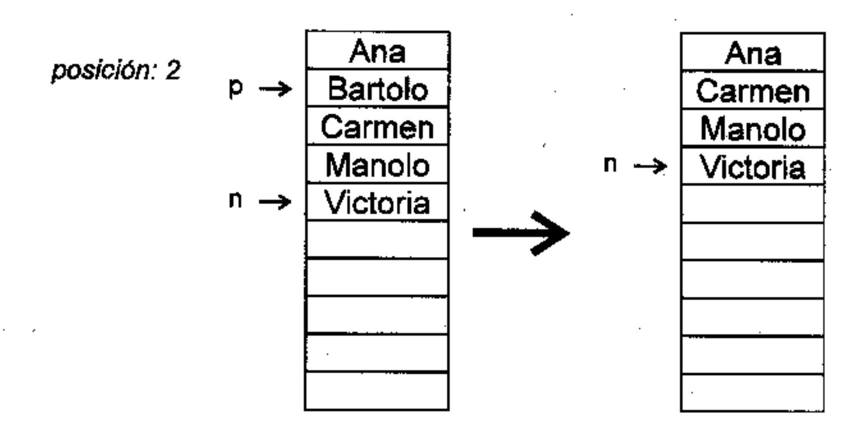
```
procedimiento InsertarEnLista( E/S vector :lista  ; E/S entero n  ;
                               S lógico : error ;
                               E entero : p  ; E cadena : e)
```

```
var
  entero : i
inicio
  // consideramos como error el que la lista esté llena o
  // o que la posición a insertar mas allá del número
  // de elementos + 1
  error ← (n = MáxLista) o (p > n+1)
  si no error entonces
    desde i ← n hasta p-1 incremento -1 hacer
      lista[i+1] ← lista[i]
    fin_desde
    lista[p] ← e
    n ← n + 1
  fin_si
fin_procedimiento
```

6.29. Disponemos de un array unidimensional de MáxLista elementos de tipo entero. Se desea diseñar un procedimiento que elimine un elemento del array situado en una posición determinada que pasamos como parámetro, conservando el array en el mismo orden.

Análisis del problema

Para borrar un elemento es necesario subir todos los componentes de la lista, desde el lugar del elemento a borrar hasta la posición `n-1`. El número de elementos ocupados del array disminuirá en una unidad. El procedimiento detectará dos condiciones de error: que el número de elementos sea 0, o que la posición a borrar sea mayor que el número de elementos.



Diseño del algoritmo

```
procedimiento BorrarDeLista(E/S vector : lista ; E/S entero  : n ;
                            S lógico : error ;E entero  :p)
var
   entero : i
inicio
   error ← (n = 0) o (p > n)
   si no error entonces
      desde i ← p hasta n hacer
         lista[i]  ← lista[i+1]
      fin_desde
      n ← n - 1
   fin_si
fin_procedimiento
```

6.30. *Algoritmo que triangule una matriz cuadrada y halle su determinante. En las matrices cuadradas el valor del determinante coincide con el producto de los elementos de la diagonal de la matriz triangulada, multiplicado por -1 tantas veces como hayamos intercambiado filas al triangular la matriz.*

Análisis del problema

El proceso de triangulación y cálculo del determinante tiene las siguientes operaciones:

- Inicializar signo a `1`
- desde `i` igual a 1 hasta `m-1` hacer:
  a) Si el elemento de lugar `(i,i)` es nulo, intercambiar filas hasta que dicho elemento sea no nulo o agotar los posibles intercambios. Cada vez que se intercambia se multiplica por -1 el valor de la variable `signo`.
  b) A continuación se busca el primer elemento no nulo de la fila *i-ésima* y, en el caso de existir, se usa para hacer ceros en la columna de abajo.

  Sea dicho elemento matriz[i,r]

  Multiplicar fila i por matriz[i+1,r]/matriz[i,r] y restarlo a la i+1

  Multiplicar fila i por matriz[i+2,r]/matriz[i,r] y restarlo a la i+2

  ....

  Multiplicar fila i por matriz[m,r]/matriz[i,r] y restarlo a la m

  Se deberá almacenar en una variable auxiliar, cs, el contenido de matriz[i+x, r], para que, en las siguientes operaciones con la fila, no afecte el resultado de

matriz[i+x, r+y] ← matriz[i+x, r+y] - matriz[i, r+y] * (cs / matriz[i, r])

a

matriz[i+x, r] ← matriz[i+x, r] - matriz[i, r] * (matriz[i+x, r] / matriz[i, r])

- Asignar al determinante el valor de signo por el producto de los elementos de la diagonal de la matriz triangulada.

Diseño del algoritmo

```
algoritmo  ejercicio_6_30
const
      m = <expresión>      // En este caso m y n serán iguales
      n = <expresión>
tipo
      array[1..m, 1..n] de real : arr
var
      arr  : matriz
      real : dt
inicio
   llamar_a leer_matriz(matriz)
   llamar_a triangula(matriz, dt)
   escribir('Determinante= ', dt)
fin

procedimiento leer_matriz (S arr : matriz)
var
   entero: i,j
inicio
   escribir('Deme los valores para la matriz')
   desde i  ←  1 hasta m hacer
      desde j ← 1 hasta n hacer
         leer( matriz[i, j])
      fin_desde
   fin_desde
fin_procedimiento

procedimiento escribir_matriz (E arr : matriz)
var
   entero   : i,j
   carácter : c
inicio
   escribir('Matriz triangulada')
   desde i ← 1 hasta m hacer
      desde j ← 1 hasta n hacer
      escribir( matriz[i, j])   // Escribir en la misma línea
      fin_desde
      escribir                  // Pasar a la siguiente línea
   fin_desde
```

```
   escribir('Pulse tecla para continuar')
   leer(c)
fin_procedimiento

procedimiento interc(E/S real: a,b)
var
   real : auxi
inicio
   auxi ← a
   a ← b
   b ← auxi
fin_procedimiento

procedimiento triangula (E arr : matriz ;  S real d: t)
var
   entero: signo
   entero: t,r,i,j
   real  : cs
inicio
   signo ← 1
   desde i ← 1 hasta m - 1 hacer
      t ← 1
      si matriz[i, i] = 0 entonces
         repetir
            si matriz[i + t, i] <> 0 entonces
               signo ←  signo * (-1)
               desde j ←  1 hasta n hacer
                  llamar_a interc(matriz[i,j], matriz[i + t,j])
               fin_desde
               llamar_a  escribir_matriz(matriz)
            fin_si
            t ←  t + 1
            hasta_que (matriz[i, i] <> 0) o (t = m - i + 1)
      fin_si
      r ← i - 1
      repetir
         r ← r + 1
      hasta_que (matriz[i, r] <> 0) o (r = n)
      si matriz[i, r] <> 0 entonces
         desde t ← i + 1 hasta m hacer
            si matriz[t, r] <> 0 entonces
               cs ←  matriz[t, r]
               desde j ←  r hasta n hacer
                  matriz[t,j]← matriz[t, j]-matriz[i, j] * (cs/matriz[i, r])
               fin_desde
               llamar_a  escribir_matriz(matriz)
            fin_si
         fin_desde
      fin_si
   fin_desde
```

```
   dt ← signo
   desde i ← 1 hasta m hacer
      dt ← dt * matriz[i, i]
   fin_desde
fin_procedimiento
```

6.31. Se desean almacenar los datos de un producto en un registro. Cada producto debe guardar información concerniente a su Código de Producto, Nombre, y Precio. Diseñar la estructura de datos correspondiente y un procedimiento que permita cargar los datos de un registro.

Análisis del problema

La estructura de datos del registro utilizado va a tener tres campos simples: `Código`, que será una cadena de caracteres, `Nombre`, que será otra cadena, y `Precio`, que será un número entero. Como se utiliza un procedimiento de lectura, es necesario crear un tipo de dato `Producto`, que será el tipo de dato utilizado como parámetro al subprograma.

Diseño del algoritmo

```
tipo
   registro : producto
      cadena : Código , Nombre
      entero : Precio
   fin
var
   producto : p
...

procedimiento LeerProducto( S producto : p)
inicio
   leer(p.Código)
   leer(p.Nombre)
   leer(p.Precio)
fin_procedimiento
```

6.32. Una farmacia desea almacenar sus productos en una estructura de registros. Cada registro tiene los campos Código, Nombre, Descripción del Medicamento (antibiótico, analgésico, etc.), Laboratorio, Proveedor, Precio, Porcentaje de IVA, Stock y Fecha de Caducidad. La fecha deberá guardar por separado el día, mes y año. Diseñe la estructura de datos y un procedimiento que permita escribir los datos de un medicamento.

Análisis del problema

Es necesario un tipo de dato con el registro, ya que será utilizado como parámetro de un procedimiento. La estructura del tipo de registro tendrá los siguientes campos: `Código` de tipo cadena, `Nombre` de tipo cadena, `Descripción` también de tipo cadena, `Precio` de tipo entero, `IVA` que indica el porcentaje del IVA a aplicar en formato `real` (por ejemplo, 0.10 para un 10%), `Stock` de tipo entero y `Caducidad`. `Caducidad` será a su vez otro registro que tendrá los campos `dd`, `mm` y `aa` que serán los tres de tipo entero. La definición de este último campo puede hacerse de dos formas; por una parte se puede definir un tipo de registro independiente, por ejemplo `Fecha` y luego hacer referencia a él en la definición del dato `Caducidad`; también se podría definir el registro `Caducidad` dentro del registro principal, utilizando lo que se llaman registros anidados.

Análisis del algoritmo

```
tipo
   registro : Fecha
      entero :dd, mm, aa
   fin
   registro : Medicamento registro
      cadena : Código , Nombre, Descripción
      entero : Precio
      real   : IVA
      entero : Stock
      Fecha  : Caducidad
   fin
var
   Medicamento : m
```

La declaración del tipo `Medicamento` también se podría hacerde la siguiente forma sin necesidad de crear el tipo `Fecha`:

```
registro : Medicamento registro
   cadena : Código , Nombre, Descripción
   entero : Precio
   real   : IVA
   entero : Stock
   registro : Caducidad
   entero : dd, mm, aa
   fin
fin
```

La implementación del procedimiento de escritura quedaría como sigue:

```
procedimiento EscribirMedicamento( S Medicamento : m )
inicio
   escribir(m.Código)
   escribir(m.Nombre)
   escribir(m.Descripción)
   escribir(m.Precio)
   escribir(m.IVA)
   escribir(m.Stock)
   escribir(m.Caducidad.dd)
   escribir(m.Caducidad.mm)
   escribir(m.Caducidad.aa)
fin_procedimiento
```

6.33. *Diseñar la estructura de datos necesaria para definir un cuadrilátero utilizando coordenadas polares.*

Análisis del problema

Para definir un Punto en el plano utilizando coordenadas polares es preciso utilizar dos datos: el ángulo y el radio. Como cada inicio de línea es el final del siguiente, se puede guardar sólo un punto por línea. Un cuadrilátero está formado por cuatro líneas, por lo que se puede definir un registro `Cuadrilátero` que tendrá cuatro campos de tipo `Punto`, es decir, a su vez registros que guardan el ángulo y el radio del punto final de cada lado. Sin embargo, es preciso guardar también la posición de inicio de cada línea, por lo que además se guardará la posición de inicio de la primera línea en otro campo del registro `Cuadrilátero`. La dirección del último lado será la misma que la de inicio.

Diseño del algoritmo

```
tipo
   registro : Punto
      real : radio, ángulo
   fin
   registro : Cuadrilátero
      Punto: Inicio, Lado1, Lado2,Lado3,Lado4
   fin
```

6.34. *Una empresa tiene almacenados a sus vendedores en un registro. Por cada vendedor se guarda su DNI, Apellidos, Nombre, Zona, Sueldo Base, Ventas Mensuales, Total Anual y Comisión. Las ventas mensuales será un vector de 12 elementos que guardará las ventas realizadas en cada uno de los meses. Total Anual será la suma de las ventas mensuales del vendedor. La Comisión se calculará aplicando un porcentaje variable al Total Anual del vendedor. Dicho porcentaje variará según las ventas anuales del vendedor, según la siguiente tabla:*

Hasta de 1.500.000	*0,00%*
Más de 1.500.000 y hasta 2.150.000	*13,75%*
Más de 2.150.000 y hasta 2.900.000	*16,50%*
Más de 2.900.000 y hasta 3.350.000	*17,60%*
Más de 3.350.000	*18,85%*

Dicha tabla se habrá cargado de un archivo secuencial que contiene tanto el límite superior como el porcentaje.

Diseñar las estructuras de datos necesarias y realizar un algoritmo que permita leer los datos del empleado, calcule el Total Anual y obtenga la Comisión que se lleva el empleado mediante la tabla descrita anteriormente, que previamente se habrá tenido que cargar del archivo.

Análisis del problema

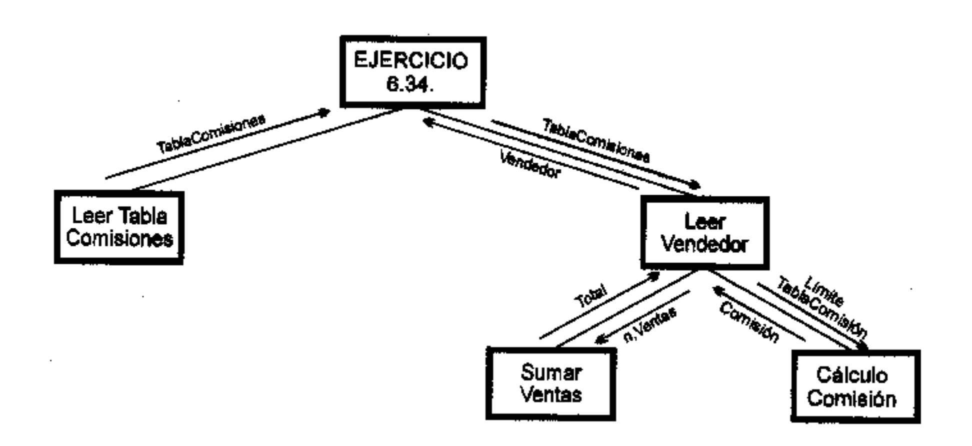
En este caso los campos del registro vendedor serán: `DNI` de tipo cadena, `Nombre` de tipo cadena, `Apellidos` de tipo cadena, `Zona` también de tipo cadena, `Sueldo` de tipo entero, `Ventas` que será un array de enteros, `Total` de tipo entero y `Comisión` que será real. Se define también un registro para la tabla de comisiones. Aunque el enunciado hable de un límite inferior y otro superior, la tabla sólo va a tener dos campos: `límite` de tipo entero y `Porcentaje` que será real.

Para realizar el cálculo de la comisión se han de cargar previamente los datos de la tabla en un array. Aunque lo más corriente sería mantener los datos de la tabla en un archivo, vamos a realizar la lectura de los datos desde teclado antes de la llamada al procedimiento que carga los datos del vendedor. Hacer la lectura antes del procedimiento y no desde dentro de él, agilizará la operación de lectura del registro si hay que leer más de uno. En dicho procedimiento, para optimizar la búsqueda de los datos se obliga a que cada límite introducido sea mayor que el anterior. También se podrían haber introducido todos los datos y ordenar el vector con algunos de los métodos que se abordarán en el capítulo siguiente.

El procedimiento de lectura leerá los campos `DNI`, `Apellidos`, `Nombre`, `Zona` y `Sueldo`. Para leer las ventas se debe utilizar un bucle que lea cada uno de los elementos de la tabla. Para calcular el total también es necesario recorrer el campo `Ventas` acumulando el valor de cada uno de los elementos. Para ello se utiliza una función `SumarVentas` que suma los elementos de un vector de enteros de `n` posiciones.

La comisión se calculará utilizando la función `CálculoComisión`. Irá analizando secuencialmente los elementos de la tabla. Como los límites se han introducido de menor a mayor, cuando se encuentre un límite mayor que el total de las ventas, se habrá encontrado la posición donde está el porcentaje de comisión a aplicar.

Por lo tanto, la arquitectura de nuestro algoritmo podría quedar de la forma siguiente:



Diseño del algoritmo

```
algoritmo ejercicio_6_35
tipo
   array[1..12] de entero : vector
   registro : RegistroComisiones
      entero : Límite, Porcentaje
   fin
   registro : Vendedores
      cadena : DNI, Apellidos, Nombre, Zona
      vector : Ventas
      entero : Sueldo, Total, Comisión
   fin
   array[1..5] de RegistroComisiones : Comisiones
var
   Comisiones : TablaComisiones
   Vendedores : Vendedor
inicio
   LeerTablaComisiones( TablaComisiones )
   LeerVendedor( TablaComisiones, Vendedor )
fin

procedimiento LeerTablaComisiones( S Comisiones : T )
var
   entero : i
inicio
   desde i ← 1 hasta 5 hacer
      // Por razones de facilidad de búsqueda, el límite que deberemos
      // introducir será el inmediato superior al fijado en la tabla
      // descrita  más arriba. Los valores a introducir serían:
      //             Límite                Porcentaje
```

```
   //            ---------------------------
   //            1500001              0.00
   //            2150001             13.75
   //            2900001             16.50
   //            3350001             17.60
   //            9999999             18.85
   // Suponiendo que la venta máxima sea de 9.999.999
   leer(T.Límite)
   leer(T.Porcentaje)
  fin_desde
fin_procedimiento

procedimiento LeerVendedor( E Comisiones : T; S Vendedores : v )
var
  entero : i
inicio
  leer(v.DNI)
  leer(v.Apellidos)
  leer(v.Nombre)
  leer(v.Zona)
  leer(v.Sueldo)
  desde i ← 1 hasta 12 hacer
    leer(v.Ventas[i])
  fin_desde
  v.Total ← SumarVentas(v.Ventas,12)
  v.Comisión ← CálculoComisión(T,v.Total)
fin_procedimiento

entero función SumarVentas( E vector : v ; E entero : n )
var
  entero : i, total
inicio
  total ← 0
  desde i ← 1 hasta 12 hacer
    total ← total + v[i]
  fin_desde
  devolver(total)
fin_función

real función CálculoComisión( E Comisiones : T ; E entero : total )
var
  entero : i
inicio
  i ← 1
  mientras T[i].Límite < total y i <> 5 hacer
    i ← i + 1
  fin_mientras
  //Si las ventas son mayores que el límite máximo (9.999.999)
  // saldríamos del bucle por la segunda condición, y i valdría 5
  devolver(total * T[i].Porcentaje)
```

```
fin_función
```

6.36. Podemos definir un polígono definiendo las coordenadas de cada uno de sus lados. Diseñar la estructura de datos que permita definir un polígono de lado n –con un máximo de 30 lados– y crear un algoritmo que permita introducir las coordenadas cartesianas de cada uno de sus lados.

Análisis del problema

Para definir un polígono de n lados, hay que definir cada una de las líneas que lo forman. Para definir una línea utilizando coordenadas cartesianas, son necesarias dos parejas de valores `x`, `y` que marcarán el inicio y el fin de cada línea en el plano. Por lo tanto se han de definir las siguientes estructuras de datos:

- `Punto` para almacenar la situación de un punto en el plano.
- `Línea`, formado una pareja de puntos que conforman el inicio y fin de la línea.
- `Polígono` que será un array de líneas. El número de elementos del array será de 30, ya que este es el número máximo que indica el enunciado. De estos 30 elementos, sólo se rellenarán `n`, que será un parámetro pasado al procedimiento.

Para introducir los datos del polígono el procedimiento recibirá como parámetro de entrada el número de lados (`n`), que debe ser siempre mayor que 2, y devolverá el array de líneas como parámetro de salida. El procedimiento irá preguntando los valores de cada uno de los puntos que configuran el polígono, teniendo en cuenta que el fin de una línea será el comienzo de otra. El final de la última línea del polígono será el primer punto introducido.

Diseño del algoritmo

```
const
  MáxLados = 30  // Número máximo de lados especificado en el enunciado
tipo
  registro : Punto
    entero : x, y
  fin
  registro : Línea
    Punto: Inicio, fin
  registro
  array[1..30] de Línea : Polígono
var
  Polígono : P
  entero : n
  ...
```

```
procedimiento LeerPolígono( S Polígono : P ; E entero :n)
var
  entero : i
inicio
  i ← 1
  LeerPunto(P[i].Inicio)
  repetir
    LeerPunto(P[i].Fin)
    i ← i + 1
    P[i].Inicio ← P[i-1].Fin
  hasta_que i = n
  P[n].Fin C P[1].Inicio
fin_procedimiento

procedimiento LeerPunto( S Punto : p)
inicio
  leer(p.x)
  leer(p.y)
fin_procedimiento
```